In this chapter we investigate the power of algorithms to solve problems. Some can be solved algorithmically and some cannot.

Why we study unsolvability:

- Useful because then can realize that searching for an algorithmic solution is a waste of time.
  - Perhaps the problem can be simplified.
- Gain a perspective on computability and its limits.
- In my view also related to complexity (Chapter 7).
  - First we study whether there is an algorithmic solution and then we study whether there is an “efficient” (polynomial-time) one.
Chapter 4.1

Decidable Languages
Decidable Languages

- We start with problems that are decidable
  - We first look at problems concerning regular languages and then those for context-free languages
Decidable Problems for Regular Languages

- We give algorithms for testing whether a finite automaton accepts a string, whether the language of a finite automaton is empty, and whether two finite automata are equivalent.

- We represent the problems by languages (not FAs).
  - Let $A_{DFA} = \{(B, w) \mid B$ is a DFA that accepts string $w\}$
  - The problem of testing whether a DFA $B$ accepts a specific input $w$ is the same as testing whether $(B, w)$ is a member of the language $A_{DFA}$.
  - Showing that the language is decidable is the same thing as showing that the computational problem is decidable.
  - So do you understand what $A_{DFA}$ represents? If you had to list the elements of $A_{DFA}$ what would they be?
A_{DFA} is a Decidable Language

- Theorem: $A_{DFA}$ is a decidable language
- Proof Idea: Present a TM $M$ that decides $A_{DFA}$
  - $M =$ On input $(B,w)$, where $B$ is a DFA and $w$ is a string:
    1. Simulate $B$ on input $w$
    2. If the simulation ends in an accept state, then accept; else reject
Outline of Proof

- Must take B as input, described as a string, and then simulate it
  - This means the algorithm for simulating any DFA must be embodied in the TM’s state transitions
  - Think about this. Given a current state and input symbol, scan the tape for the encoded transition function and then use that info to determine new state

- The actual proof would describe how a TM simulates a DFA
  - Can assume B is represented by its 5 components and then we have w
    - Note that the TM must be able to handle any DFA, not just this one
  - Keep track of current state and position in w by writing on the tape
    - Initially current state is q0 and current position is leftmost symbol of w
  - The states and position are updated using the transition function $\delta$
    - TM M’s $\delta$ not the same as DFA B’s $\delta$
  - When M finishes processing, accept if in an accept state; else reject. The implementation will make it clear that will complete in finite time.
A_NFA is a Decidable Language

Proof Idea:

Because we have proven decidability for DFAs, all we need to do is convert the NFA to a DFA.

N = On input (B,w) where B is an NFA and w is a string

1. Convert NFA B to an equivalent DFA C, using the procedure for conversion given in Theorem 1.39
2. Run TM M on input (C,w) using the theorem we just proved
3. If M accepts, then accept; else reject

Running TM M in step 2 means incorporating M into the design of N as a subroutine

Note that these proofs allow the TM to be described at the highest of the 3 levels we discussed in Chapter 3 (and even then, without most of the details!).
Computing whether a DFA accepts any String

- \( E_{\text{DFA}} = \{ <A> | A \text{ is a DFA and } L(A) = \emptyset \} \) is a decidable language
- **Proof:**
  - A DFA accepts some string iff it is possible to reach the accept state from the start state. How can we check this?
  - We can use a marking algorithm similar to the one used in Chapter 3.
  - \( T = \text{On input (A) where A is a DFA:} \)
    1. Mark the start state of A
    2. Repeat until no new states get marked:
      3. Mark any state that has a transition coming into it from any state already marked
    4. If no accept state is marked, accept; otherwise reject
  - In my opinion this proof is clearer than most of the previous ones because the pseudo-code above specifies enough details to make it clear how to implement it
EQ_{DFA} is a Decidable Language

- EQ_{DFA} = \{(A,B) | A and B are DFAs and L(A) = L(B)\}
- Proof idea
  - Construct a DFA C from A and B, where C accepts only those strings accepted by either A or B but not both (symmetric difference)
  - If A and B accept the same language, then C will accept nothing and we can use the previous proof (for E_{DFA}) to check for this.
  - So, the proof is:
    - F = On input (A,B) where A and B are DFAs:
      1. Construct DFA C that is the symmetric difference of A and B (details on how to do this on next slide)
      2. Run TM T from the proof from last slide on input (C)
      3. If T accepts (sym. diff = ∅) then accept. If T rejects then reject
How to Construct C

- \[ L(C) = (L(A) \cap L(B))' \cup (L(A)' \cap L(B)) \]

- We used proofs by construction that regular languages are closed under \( \cup, \cap, \) and complement.

- We can use those constructions to construct a FA that accepts \( L(C) \)

  - Wait a minute! The book is quite cavalier! We never proved regular languages are closed under \( \cap \)
Regular Languages Closed under Intersection

- If $L$ and $M$ are regular languages, then so is $L \cap M$

- **Proof:** Let $A$ and $B$ be DFAs whose regular languages are $L$ and $M$, respectively.

- Construct $C$, the “product automation” of $A$ and $B$
  - More on this in a minute, but essentially $C$ tracks the states in $A$ and $B$ (just like when we did the proof of union without using NFAs).

- Make the final states of $C$ be the pairs consisting of final states of both $A$ and $B$
  - In the union case we the final state any state with a final state in $A$ or $B$. 

12
Example: Product DFA for Intersection
A\textsubscript{CFG} is a Decidable Language

**Proof Idea:**

- For CFG G and string w want to determine whether G generates w. One idea is to use G to go through all derivations. This will not work, why?
  - Because this method a best will yield a TM that is a recognizer, not a decider. Can generate infinite strings and if not in the language, will never know it.
  - But since we know the length of w, we can exploit this. How?
  - A string w of length n will have a derivation that uses 2n-1 steps if the CFG is in Chomsky-Normal Form.
    - So first convert to Chomsky-Normal Form
    - Then list all derivations of length 2n-1 steps. If any generates w, then accept, else reject.
    - This is a variant of breadth first search, but instead of extended the depth 1 at a time we allow it to go 2n-1 at a time. As long as finite depth extension, we are okay
E_{CFG} is a Decidable Language

How can you do this? What is the brute force approach?

- Try all possible strings w. Will this work?
  - The number is not bounded, so this would not be decidable
  - Instead, think of this as a graph problem where you want to know if you can reach a string of terminals from the start state

- Do you think it is easier to work forward or backwards?
  - Answer: backwards
E_{CFG} is a Decidable Language (cont)

- **Proof Idea:**
  - Can the start variable generate a string of terminals?
  - Determine for each variable if it can generate any string of terminals and if so, mark it
  - Keep working backwards so that if the right-side of any rule has only marked items, then mark the LHS
    - For example, if X \rightarrow YZ and Y and Z are marked, then mark X
    - If you mark S, then done; if nothing else to mark and S not marked, then reject
    - You start by marking all terminal symbols
EQ_{CFG} is not a Decidable Language

- We cannot reuse the reasoning to show that EQ_{DFA} is a decidable language since CFGs are not closed under complement and intersection.
- As it turns out, EQ_{CFG} is not decidable!
- We will learn in Chapter 5 how to prove things undecidable.
Every Context-Free Language is Decidable

- Note that a few slides back we showed $A_{CFG}$ is decidable.
- This is almost the same thing
- We want to know if $A$, which is a CFL, is decidable.
  - $A$ will have some CFG $G$ that generates it
  - When we proved that $A_{CFG}$ is decidable, we constructed a TM $S$ that would tell us if any CFG accepts a particular input $w$.
  - Now we use this TM and run it on input $<G,w>$ and if it accepts, we accept, and if it rejects, we reject.
- This is so close to the prior proof it is confusing. It comes from the fact that a CFL is defined by a CFG.
- This leads us to the following picture of the hierarchy of languages
We proved Regular $\subseteq$ Context-free since we can convert a FA into a CFG.

We just proved that every Context-free language is decidable.

From the definitions in Chapter 3 it is clear that every Decidable language is trivially Turing-recognizable. We hinted that not every Turing-recognizable language is Decidable. Next we prove that!
Chapter 4.2

The Halting Problem
The Halting Problem

- One of the most philosophically important theorems in the theory of computation
  - There is a specific problem that is algorithmically unsolvable.
  - In fact, ordinary/practical problems may be unsolvable
    - Software verification
      - Given a computer program and a precise specification of what the program is supposed to do (e.g., sort a list of numbers)
      - Come up with an algorithm to prove the program works as required
        - This cannot be done!
        - But wait, can’t we prove a sorting algorithm works?
        - Note: the input has two parts: specification and task. The proof is not only to prove it works for a specific task, like sorting numbers.

- Our first undecidable problem:
  - Does a TM accept a given input string?
    - Note: we have shown that a CFL is decidable and a CFG can be simulated by a TM. This does not yield a contradiction. TMs are more expressive than CFGs.
Halting Problem II

- $A_{TM} = \{(M,w) \mid M \text{ is a TM and } M \text{ accepts } w\}$
- $A_{TM}$ is undecidable
  - It can only be undecidable due to a loop of $M$ on $w$.
  - If we could determine if it will loop forever, then could reject. Hence $A_{TM}$ is often called the halting problem.
    - As we will show, it is impossible to determine if a TM will always halt (i.e., on every possible input).
  - Note that this is Turing recognizable:
    - Simulate $M$ on input $w$ and if it accept, then accept; if it ever rejects, then reject
  - We start with the diagonalization method
In 1873 mathematician Cantor was concerned with the problem of measuring the sizes of infinite sets.

How can we tell if one infinite set is bigger than another or if they are the same size?

- We cannot use the counting method that we would use for finite sets. Example: how many even integers are there?
- What is larger: the set of even integers or the set of all strings over \{0,1\} (which is the set of all integers)

Cantor observed that two finite sets have the same size if each element in one set can be paired with the element in the other

- This can work for infinite sets
Function Property Definitions

From basic discrete math (e.g., CS 1100)

- Given a set A and B and a function $f$ from A to B
  - $f$ is one-to-one if it never maps two elements in A to the same element in B
    - The function \textit{add-two} is one-to-one whereas \textit{absolute-value} is not
  - $f$ is onto if every item in B is reached from some value in a (i.e., $f(a) = b$ for every $b \in B$).
    - For example, if A and B are the set of integers, then \textit{add-two} is onto but if A and B are the positive integers, then it is not onto since $b = 1$ is never hit.
  - A function that is one-to-one and onto has a (one-to-one) correspondence
    - This allows all items in each set to be paired
An Example of Pairing Set Items

- Let \( N \) be the set of natural numbers \( \{1, 2, 3, \ldots\} \) and let \( E \) be the set of even natural numbers \( \{2, 4, 6, \ldots\} \).

- Using Cantor’s definition of size we can see that \( N \) and \( E \) have the same size.
  - The correspondence \( f \) from \( N \) to \( E \) is \( f(n) = 2n \).

- This may seem bizarre since \( E \) is a proper subset of \( N \), but it is possible to pair all items, since \( f(n) \) is a 1:1 correspondence, so we say they are the same size.

- Definition:
  - A set is *countable* if either it is finite or it has the same size as \( N \), the set of natural numbers.
Example: Rational Numbers

- Let \( Q = \{m/n: m,n \in \mathbb{N}\} \), the set of positive Rational Numbers

- \( Q \) seems much larger than \( \mathbb{N} \), but according to our definition, they are the same size.
  - Here is the 1:1 correspondence between \( Q \) and \( \mathbb{N} \)
  - We need to list all of the elements of \( Q \) and then label the first with 1, the second with 2, etc.
    - We need to make sure each element in \( Q \) is listed only once
### Correspondence between N and Q

- To get our list, we make an infinite matrix containing all the positive rational numbers.
  - Bad way is to make the list by going row-to-row. Since 1\textsuperscript{st} row is infinite, would never get to the second row.
  - Instead use the diagonals, not adding the values that are equivalent.
    - So the order is 1/1, 2/1, ½, 3/1, 1/3, …
  - This yields a correspondence between Q and N.
    - That is, N=1 corresponds to 1/1, N=2 corresponds to 2/1, N=3 corresponds to ½ etc.
Theorem: R is Uncountable

- A real number is one that has a decimal representation and R is set of Real Numbers
  - Includes those that cannot be represented with a finite number of digits, like Pi and square root of 2
- Will show that there can be no pairing of elements between R and N
  - Will find some x that is always not in the pairings and thus a proof by contradiction
Finding a New Value $x$

- To the right is an example mapping
  - Assume that it is complete
- I now describe a method that will be guaranteed to generate a value $x$ not already in the infinite list
- Generate $x$ to be a real number between 0 and 1 as follows
  - To ensure that $x \neq f(1)$, pick a digit not equal to the first digit after the decimal point. Any value not equal to 1 will work. Pick 4 so we have .4
  - To $x \neq f(2)$, pick a digit not equal to the second digit. Any value not equal to 5 will work. Pick 6. We have .46
  - Continue, choosing values along the “diagonal” of digits (i.e., if we took the $f(n)$ column and put one digit in each column of a new table).
- When done, we are guaranteed to have a value $x$ not already in the list since it differs in at least one position with every other number in the list.

<table>
<thead>
<tr>
<th>$n$</th>
<th>$f(n)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3.14159…</td>
</tr>
<tr>
<td>2</td>
<td>55.5555…</td>
</tr>
<tr>
<td>3</td>
<td>0.12345…</td>
</tr>
<tr>
<td>4</td>
<td>0.500000</td>
</tr>
</tbody>
</table>
Implications

- The theorem we just proved about $\mathbb{R}$ being uncountable has an important application in the theory of computation.
  - It shows that some languages are not decidable or even Turing-recognizable, because there are uncountably many languages yet only countably many Turing Machines.
    - Because each Turing machine can recognize a single language and there are more languages than Turing machines, some languages are not recognized by any Turing machine.
      - Corollary: some languages are not Turing-recognizable.
Some Languages are Not Turing-recognizable I

- The set of all strings $\Sigma^*$ is countable
  - A finite number of strings of each length, so we can list them by increasing length and hence they are countable

- The set of all Turing Machines $M$ is countable since each TM $M$ has an encoding into a string $\langle M \rangle$
  - Order by length and omit strings that do not represent valid TM’s and we have a countable list of Turing Machines
Some Languages are Not Turing-recognizable II

- The set of all languages L over \( \Sigma \) is uncountable
  - Recall a language is made up of a set of strings, so different from what we just counted on last slide.
  - Each language is represented by an infinite binary sequence B, where each position in the sequence corresponds to a string
    - Assume \( \Sigma^* = \{s_1, s_2, s_3 \ldots \} \). We can encode any language as a characteristic binary sequence, where the bit indicates whether the corresponding \( s_i \) is a member of the language. Thus, there is a 1:1 mapping.
    - The set of all infinite binary sequences B is uncountable
    - Can prove uncountable using same proof used to prove real numbers not countable
  - L is uncountable because it has a correspondence with B
  - Since B is uncountable and L and B are of equal size, L is uncountable

- So set of TMs is countable and the set of languages is not
  - Means we cannot put set of languages into a correspondence with set of TMs.
  - Therefore some languages do not have a corresponding Turing machine
  - Thus some languages not Turing-Recognizable
Common Sense Explanation

- Comparing languages, a potentially infinite set of strings, versus number of strings
- Each language is represented by a sequence of infinite length whereas each individual string is of finite (but unbounded) length
- String is to Language as Natural number is to Real Number
Halting Problem is Undecidable

- Prove that halting problem is undecidable
  - We started this a while ago …
  - Let $A_{TM} = \{<M,w>| M \text{ is a TM and accepts } w\}$
- Proof Technique:
  - Assume $A_{TM}$ is decidable and obtain a contradiction
  - A diagonalization proof
Proof: Halting Problem is Undecidable

- Assume $A_{TM}$ is decidable
- Let $H$ be a decider for $A_{TM}$
  - On input $<M,w>$, where $M$ is a TM and $w$ is a string, $H$ halts and accepts if $M$ accepts $w$; otherwise it rejects
- Construct a TM $D$ using $H$ as a subroutine
  - $D$ calls $H$ to determine what $M$ does when input string is its own description $<M>$.
    - Like running a C++ program where input is the program represented as a string
  - $D$ then outputs the opposite of $H$’s answer
  - $D(<M>)$ accepts if $M$ does not accept $<M>$ and rejects if $M$ accepts $<M>$
- Now run $D$ on its own description
  - $D(<D>)$ = accept if $D$ does not accept $<D>$ and reject if $D$ accepts $<D>$
  - No matter what $D$ does it is forced to do the opposite, which is a contradiction. Thus, neither TM $D$ or TM $H$ can exist. See next slide.
The Diagonalization Proof

<table>
<thead>
<tr>
<th></th>
<th>&lt;M1&gt;</th>
<th>&lt;M2&gt;</th>
<th>&lt;M3&gt;</th>
<th>&lt;M4&gt;</th>
<th>...</th>
<th>&lt;D&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>M1</td>
<td>Accept</td>
<td>Reject</td>
<td>Accept</td>
<td>Reject</td>
<td>...</td>
<td>Accept</td>
</tr>
<tr>
<td>M2</td>
<td>Accept</td>
<td>Accept</td>
<td>Accept</td>
<td>Accept</td>
<td>...</td>
<td>Accept</td>
</tr>
<tr>
<td>M3</td>
<td>Reject</td>
<td>Reject</td>
<td>Reject</td>
<td>Reject</td>
<td>...</td>
<td>Reject</td>
</tr>
<tr>
<td>M4</td>
<td>Accept</td>
<td>Accept</td>
<td>Reject</td>
<td>Reject</td>
<td>...</td>
<td>Accept</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>?</td>
</tr>
</tbody>
</table>

The TM D must invert the value on the diagonal. It can do this for <M1>, <M2>, etc, but not for <D>. If the entry for D(<D>) was accept then it needs to be reject, and if it was reject then it needs to be accept. Contradiction. Similar to proof that Real numbers not countable.
A More Satisfying Proof for CS Majors

- The last proof uses some mathematical tricks and is not very intuitive

- Computer programs appear more concrete to most of us

- The halting problem naturally is about programs and infinite loops

- After teaching this many times, I developed a proof that most of you will find less arbitrary since it focuses programs and infinite loops.
  - But it is nonetheless follows the same steps as the prior proof
You write a program, `halts(P, X)` in C++ that takes as input any C++ program, P, and the input to that program, X

- Your program `halts(P, X)` analyzes P and returns “yes” if P will halt on X and “no” if P will not halt on X

You now write a short procedure `foo(X)`:  
`foo(X) {a: if halts(X,X) then goto a; else halt}`  
This program does not halt if P halts on X (infinite loop via goto) and it does if P does not halt on X

Does `foo(foo)` halt?

- It halts if and only if `halts(foo,foo)` returns no
  - It halts if and only if it does not halt. Contradiction.

Thus we have proven that you cannot write a program to determine if any arbitrary program will halt or loop
What does this mean?

- Recall what was said earlier
  - The halting problem is not some contrived problem
  - The halting problem asks whether we can tell if some TM M will accept an input string
  - We are asking if the language below is decidable
    - \( A_{TM} = \{(M,w) | M \text{ is a TM and } M \text{ accepts } w\} \)
  - It is not decidable
    - But as I keep emphasizing, M is an input variable too!
      - Of course, some algorithms are decidable, like sorting algorithms
  - Halting problem is Turing-recognizable (we discussed this)
    - Simulate the TM on w and if it accepts/rejects, then accept/reject.
  - The halting problem is special because it gets at the heart of the matter (it is related to \( A_{TM} \) in general)
A language is co-Turing recognizable if it is the complement of a Turing-recognizable language.

Theorem: A language is decidable if and only if it is Turing-recognizable and co-Turing-recognizable.

Why? To be Turing-recognizable, we must accept in finite time. If we don’t accept, we may reject or loop (in which case it is not decidable).

Since we can invert any “question” by taking the complement, taking the complement flips the accept and reject answers. Thus, if we invert the question and it is Turing-recognizable, then that means that we would get the answer to the original reject question in finite time.
More Formal Proof

- Theorem: A language is decidable iff it is Turing-recognizable and co-Turing-recognizable
- Proof (2 directions)
  - Forward direction easy. If it is decidable, then both it and its complement are Turing-recognizable
  - Other direction:
    - Assume A and A’ are Turing-recognizable and let M1 recognize A and M2 recognize A’
    - The following TM will decide A
    - M = On input w
      1. Run both M1 and M2 on input w in parallel
      2. If M1 accepts, accept; if M2 accepts, then reject
    - Every string is in either A or A’ so every string w must be accepted by either M1 or M2. Because M halts whenever M1 or M2 accepts, M always halts and so is a decider.
    - Furthermore, it accepts all strings in A and rejects all not in A, so M is also a decider for A and thus A is decidable
Implication

- For any undecidable language, either the language or its complement is not Turing-recognizable
Complement of $A_{TM}$ is not Turing-recognizable

- $A_{TM}'$ is not Turing-recognizable

Proof:

- We know that $A_{TM}$ is Turing-recognizable but not decidable
- If $A_{TM}'$ were also Turing-recognizable, then $A_{TM}$ would be decidable, which it is not
- Thus $A_{TM}'$ is not Turing-recognizable

This should not be too surprising.

- It is harder to determine that something is not in the language
Due to time constraints we are only going to cover the first 3 pages of this chapter. However, we cover the notion of reducibility in depth when we cover Chapter 7.
What is Reducibility?

- A reduction is a way of converting one problem to another such that the solution to the second can be used to solve the first.

  - We say that problem A is reducible to problem B.
  
  - Example: finding your way around NY City is reducible to the problem of finding and reading a map.

  - If A reduces to B, what can we say about the relative difficulty of problem A and B?
    
    1. A can be no harder than B since the solution to B solves A.
    2. A could be easier (the reduction is “inefficient” in a sense).
    3. In example above, A is easier than B since B can solve any routing problem.
Practice on Reducibility

- In our previous class work, did we reduce NFAs to DFAs or DFAs to NFAs?
  - We reduced NFAs to DFAs
    - We showed that an NFA can be reduced (i.e., converted) to a DFA via a set of simple steps
    - NFA can not be any more powerful than a DFA
    - Based only on the reduction, NFA could be less powerful
    - But since we know this is not possible, since an DFA is a degenerate form of an NFA, we showed they have the same expressive power
How Reducibility is used to Prove Languages Undecidable

- If A is reducible to B and B is decidable, what can we say?
  - A is decidable (since A can only be “easier”)
  - Also, B, which is decidable, can be used to solve A

- If A is reducible to B and A is decidable, what can we say?
  - Nothing—B may not be decidable (so this is not useful for us)

- If A is undecidable and reducible to B, then what can we say about B?
  - B must be undecidable (B can only be harder than A)
  - This is the most useful part for Chapter 5, since this is how we can prove a language undecidable
    - We can leverage past proofs and not start from scratch

- To show something undecidable, show an undecidable problem can be reduced to it.
Example: Prove $\text{HALT}_{\text{TM}}$ is Undecidable I

- Need to reduce $A_{\text{TM}}$ to $\text{HALT}_{\text{TM}}$, where $A_{\text{TM}}$ already proven to be undecidable
  - Can use $\text{HALT}_{\text{TM}}$ to solve $A_{\text{TM}}$

- Proof by contradiction
  - Assume $\text{HALT}_{\text{TM}}$ is decidable and show this implies $A_{\text{TM}}$ is decidable
    - Assume TM $R$ that decides $\text{HALT}_{\text{TM}}$
    - Use $R$ to construct $S$ a TM that decides $A_{\text{TM}}$
    - Pretend you are $S$ and need to decide $A_{\text{TM}}$ so if given input $<M, w>$ must output accept if $M$ accepts $w$ and reject if $M$ loops on $w$ or rejects $w$.
      - First try: simulate $M$ on $w$ and if it accepts then accept and if rejects then reject. But in trouble if it loops.
      - This is bad because we need to be a decider
Example: Prove $\text{HALT}_{\text{TM}}$ is Undecidable II

- Instead, use assumption that have TM $R$ that decides $\text{HALT}_{\text{TM}}$

- Now can test if $M$ halts on $w$
  - If $R$ indicates that $M$ does halt on $w$, you can use the simulation and output the same answer
  - If $R$ indicates that $M$ does not halt, then reject since infinite looping on $w$ means it will never accept
  - The formal solution on next slide
  - We already discussed this case when we informally discussed how the halting problem is related to $A_{\text{TM}}$
Solution: $\text{HALT}_\text{TM}$ is Undecidable

- Assume TM R decides $\text{HALT}_\text{TM}$
- Construct TM S to decide $A_\text{TM}$ as follows

$S = \text{"On input } <M, w>\text{, an encoding of a TM M and a string w:

1. Run TM R on input } <M, w>
2. If R rejects (doesn’t halt), reject
3. If R accepts, simulate M on w until it halts
4. If M has accepted, accept; If M has rejected, reject"