Chapter 6 cont...
Turing Machines

Turing's Thesis

Turing's thesis:

Any computation carried out by mechanical means can be performed by a Turing Machine

(1930)

Computer Science Law:

A computation is mechanical
if and only if
it can be performed by a Turing Machine

There is no known model of computation
more powerful than Turing Machines

Algorithms are Turing Machines

When we say:

There exists an algorithm

We mean:

There exists a Turing Machine
that executes the algorithm

Undecidability

- RE languages are accepted (recognized) by TM's.
- RE languages may be grouped into two classes:
  - Class 1 (*recursive language*) --- each language *L* in this class has a TM (thought as an algorithm) which not only accepts strings of *L*, but also tells us what strings are not in *L* by *halting*.
  - Class 2 (*RE but not recursive*) --- each language *L* in this class has a TM (*not* thought as an algorithm) which accepts strings of *L*, but may not halt when a given input string is not in *L*.

- Formally, a language L is *recursive* if $L = L(M)$ for some TM M such that:

  (1) If $w \in L$, then M accepts (and therefore halts).

  (2) If $w \notin L$, then M *eventually halts*, although it *never* enters an accepting state (i.e., "reject") .

- A TM of this type corresponds to the formal notion of *algorithm*.

Church-Turing Thesis

- A Turing machine that halts on all inputs is the precise formal notion corresponding to the intuitive notion of an algorithm.

- An "algorithm" means a precisely defined set of instructions

- This thesis cannot be formally proven

Consequence of Church-Turing Thesis:

If a problem cannot be solved by a Turing machine then it cannot be solved by a human using a precisely defined sequence of instructions

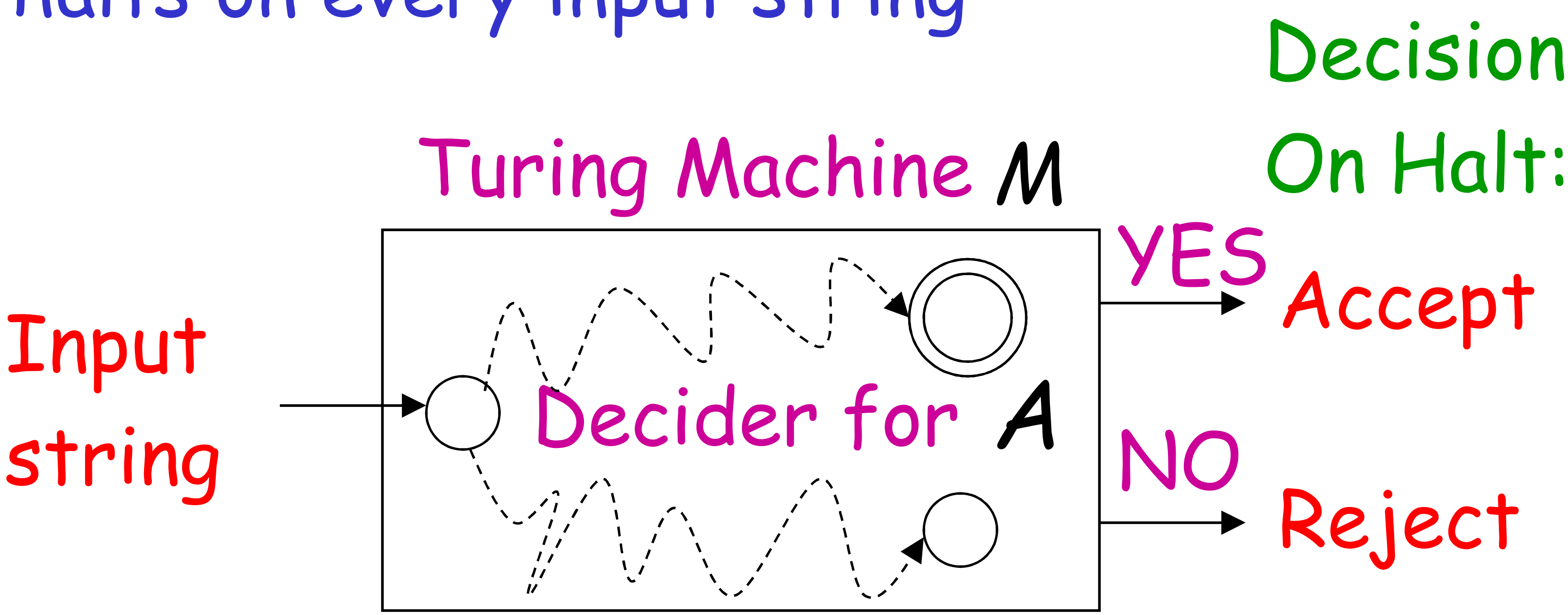
A given language *L*, *regarded as a problem*, is called *decidable* if *L* is a **recursive language**; and *undecidable* if not.

The existence or nonexistence of an algorithm to solve a problem (i.e., the problem is decidable or undecidable) is *more important* than the existence or nonexistence of a TM to solve the problem.

Decidable Languages

Recall that:

A language A is decidable,
if there is a Turing machine M (decider)
that accepts the language A and
halts on every input string



A computational problem is decidable
if the corresponding language is decidable

We also say that the problem is solvable

Undecidable Languages

undecidable language = not decidable language

There is no decider:

there is no Turing Machine
which accepts the language
and makes a decision (halts)
for every input string

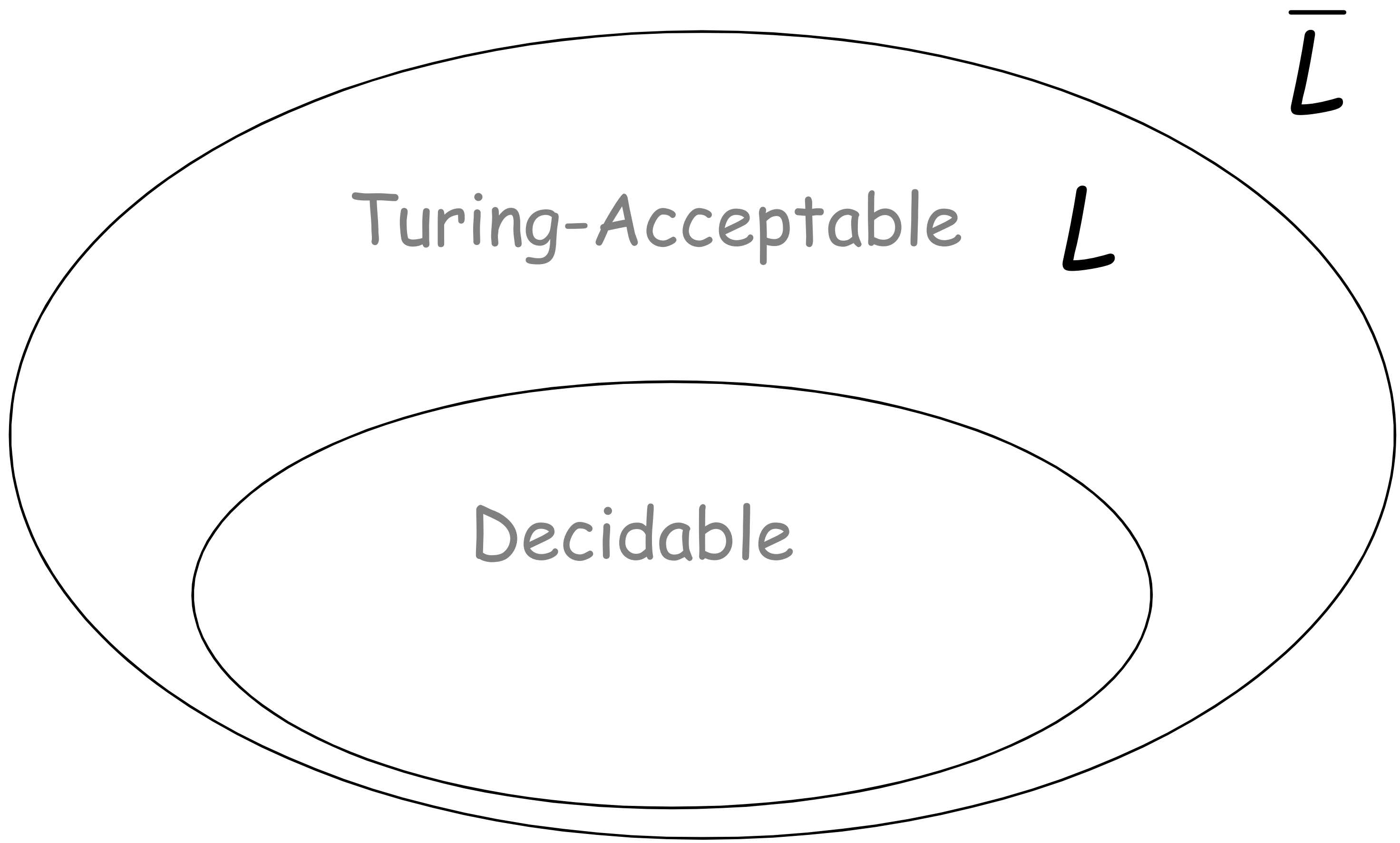
(machine may make decision for some input strings)

For an undecidable language, the corresponding problem is undecidable (unsolvable):

There is no Turing Machine (Algorithm) that gives an answer (yes or no) for every input instance

(answer may be given for some input instances)

Remember that there are undecidable languages (i.e. also undecidable problems):



L is Turing-Acceptable and undecidable

We will prove that two particular problems are unsolvable:

Membership problem

Halting problem

Proofs of *Decidability*

How can you prove a language is *decidable*?

What Decidable Means

A language L is decidable if there exists a TM M such that for all strings w:

- If $w \in L$, M enters q_{Accept}.
- If $w \notin L$, M enters q_{Reject}.

To prove a language is decidable, we can show how to construct a TM that decides it.

For a correct proof, need a convincing argument that the TM always eventually accepts or rejects any input.

Proofs of Undecidability

To prove a language is *undecidable*, need to show there is **no** Turing Machine that can decide the language.

This is hard: requires reasoning about *all* possible TMs.

Reducibility

Reducibility

- Method for proving that problems are computationally unsolvable.

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

Reduction

Let's say we want to solve problem of class A and we know how to solve problems of class B

- If for *every* problem of class A, we can find a way to convert it *some* problem of class B, ...

  then, we can solve all problems of class A using our method for solving problems of class B.

- We can also talk about how much effort is needed to **transform** the problem. For most of what we are interested in here, it is enough that the transformation can be ***computed*** by a Turing machine.

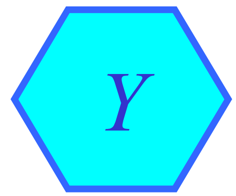
Proof by Reduction

1. We know X does not exist.

(e.g., X = a TM that can decide A_{TM})

2. Assume Y exists.

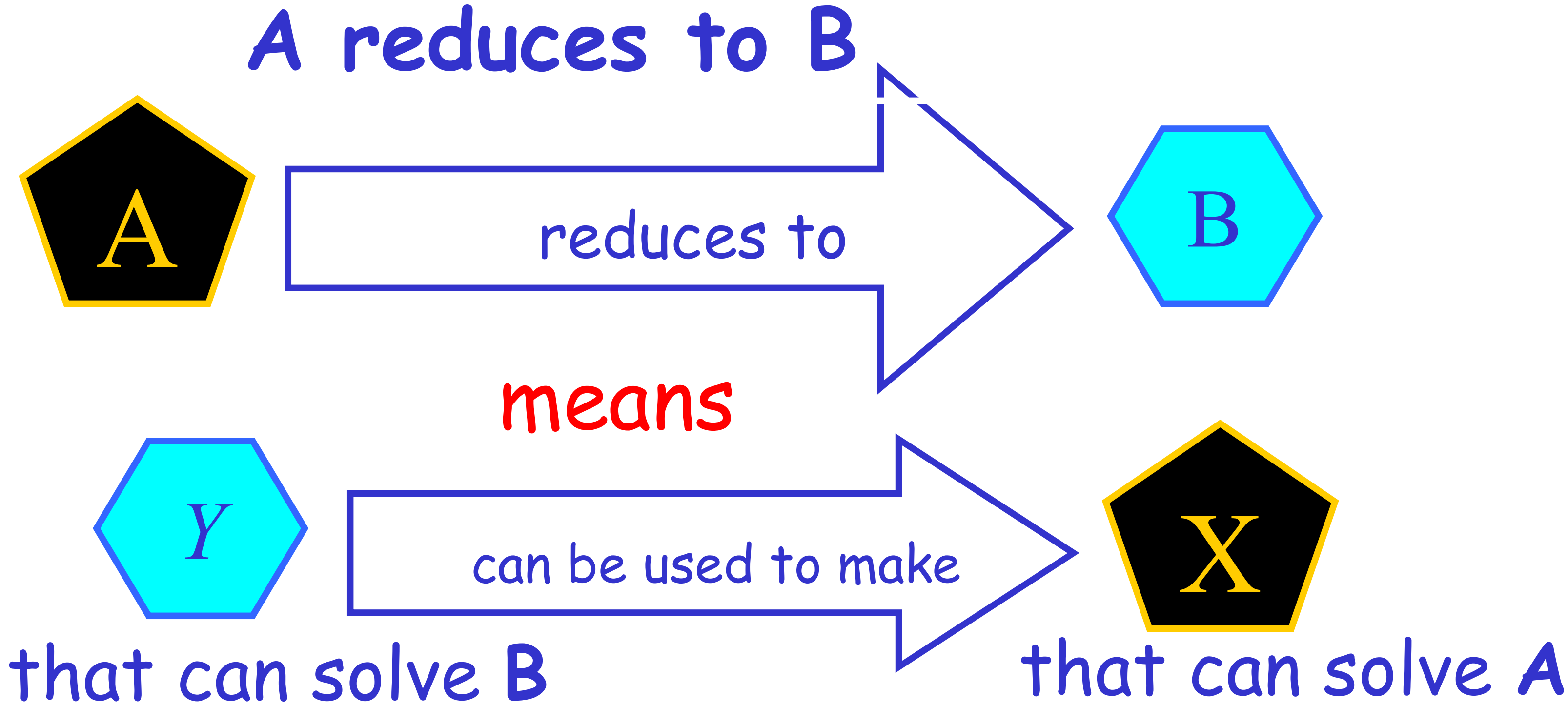
(e.g., Y = a TM that can decide B)



3. Show how to use Y to make X.

4. Since X does not exist, but Y could be used to make X, then Y must not exist.

Reduction Proofs



Hence, **A is not a harder problem than B.**

Converse?

A reduces to B

that can solve **B** that can solve ***A***

A is not a harder problem than B.

Does this mean **B** is as hard as ***A***?

No! ***Y*** can be ***any*** solver for **B**. **X** is ***one*** solver for ***A***. There might be easier solvers for ***A***.

Generally…

- Reducibility plays an important role in classifying problems by **decidability**, and later in **complexity theory** as well.
- When A is reducible to B, solving A cannot be harder than solving B because a solution to B gives a solution to A.
- In terms of computability theory, if A is reducible to B and B is decidable, A also is decidable. Equivalently, ***if A is undecidable and reducible to B, B is undecidable.***
- This last version is key to proving that various problems are undecidable.