

---

# Encodings and reducibility

---

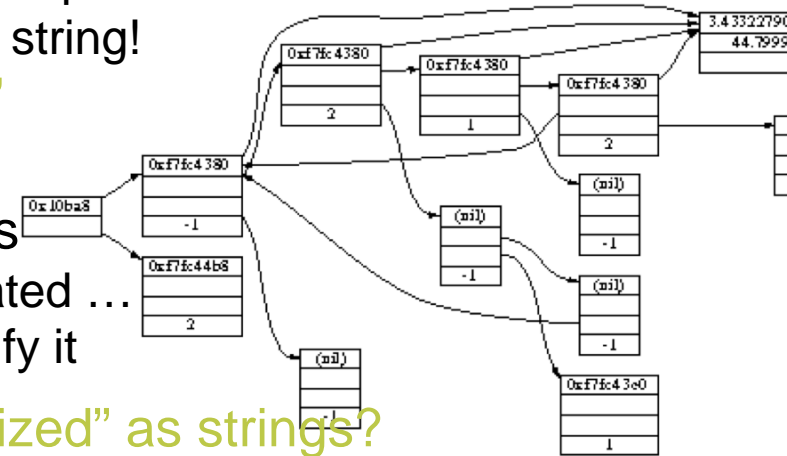
---

# Designing your little language

- You have to encode your problem as a string
  - How would you encode a tree?
    - That is, what's a nice little language for trees?
      - More than one option?
    - What solvers could you write?
    - Does every string encode some tree?
  - How would you encode an arbitrary graph?
    - Is it okay if there are two encodings of the same graph?
    - What solvers could you write?

# Remind me why we're using strings?

- Why not design a Java graph object & write solver to work on that?
  - Then your solver can only take **input** from another Java program
  - Can only send its **output** (if a graph) to another Java program
  - Where do those programs get *their* input and send *their* output?
  - All programs must be running **at once**, to see same object
    - How do you **save** or **transmit** a structure with pointers?
    - Solution is **serialization** – turn object into a string!
- Strings are a “least common denominator”
  - Simple storage
  - Simple communication between programs
    - can even peek at what's being communicated ...
    - and even run programs to analyze or modify it
- Can all finite data structures really be “serialized” as strings?
  - Sure ... computer's memory can be regarded as a *string of bytes*.
  - Theoretical CS even regards all problems as string problems!
    - A Turing machine's input and output tapes can be regarded as *strings*
    - **P** = “polynomial time” = class of decision problems that are  $O(n^k)$  for some constant  $k$ , where  $n = \text{length of the input string}$



---

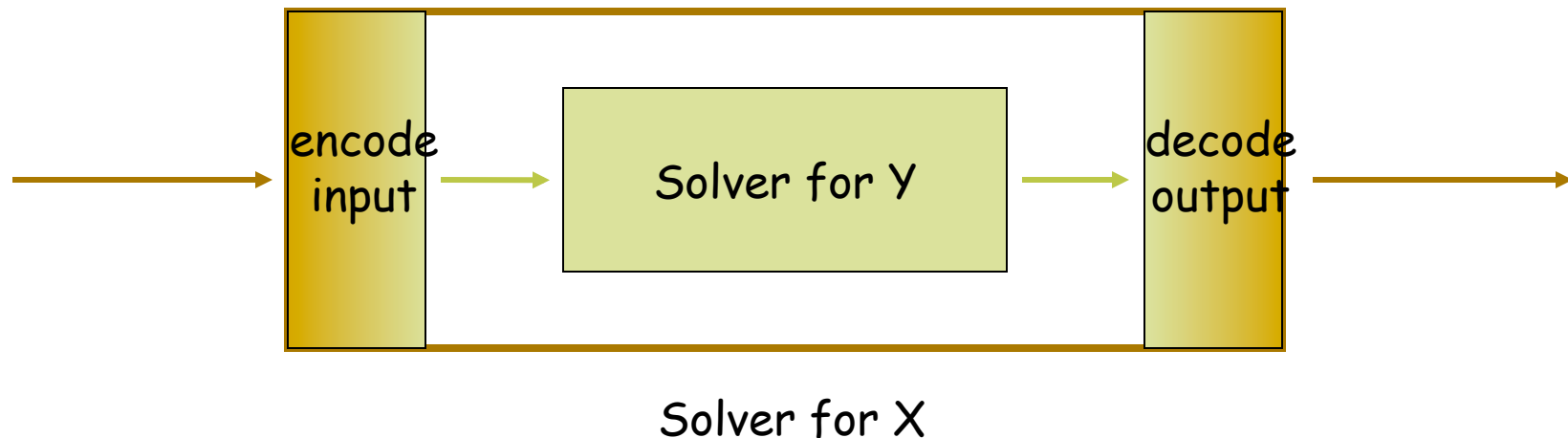
# Encoding mismatches

- How do you run graph algorithms on a string that represents a tree?
- How do you run tree algorithms on a string that represents a graph?
  - Assuming you know the graph is a tree
  - Should reject graphs that aren't trees (syntax error)
- (What's the corresponding problem & solution for Java classes?)
- How do you run a graph algorithm to sort a sequence of numbers?



# Reducibility

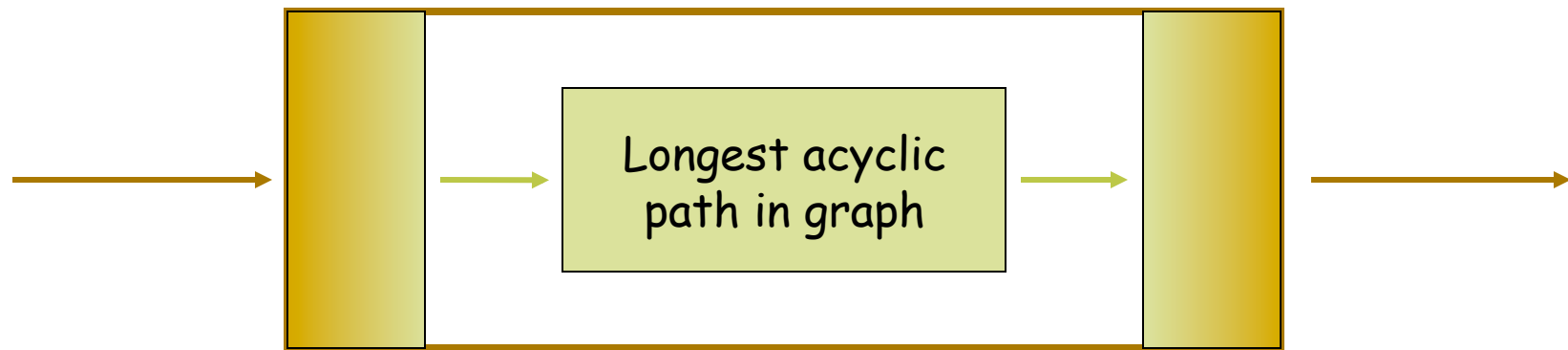
- One way to build a solver is to “wrap” another solver



$$X(\text{input}) = \text{decode}(Y(\text{encode}(\text{input})))$$

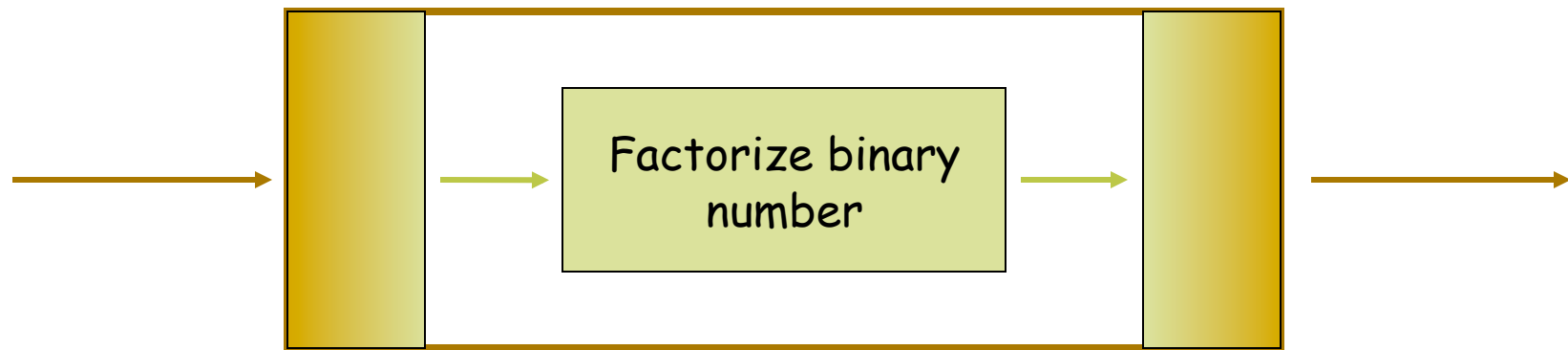
- Can set this up without knowing how to solve Y!
  - As we find (faster) solvers for Y, automatically get (faster) solvers for X

# Reducibility



Sort

# Reducibility



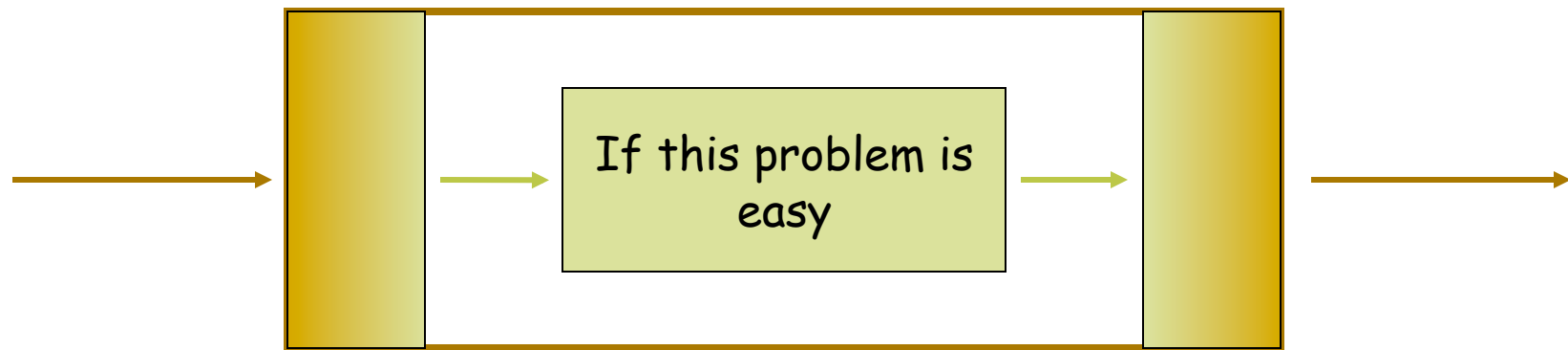
Factorize Roman numeral

# Reducibility



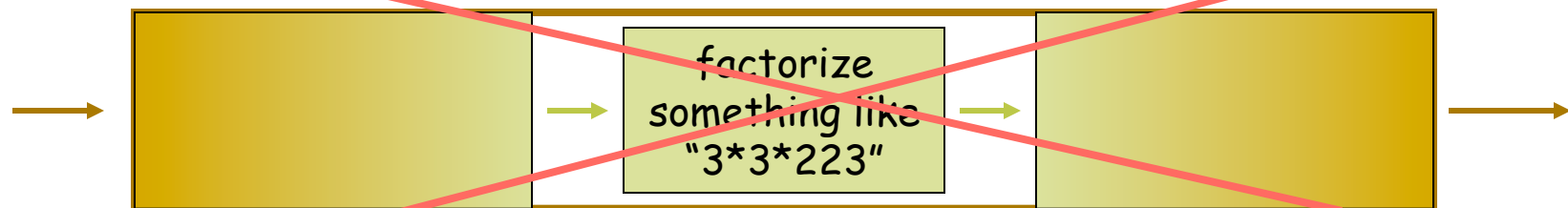
Factorize binary number

# Reducibility



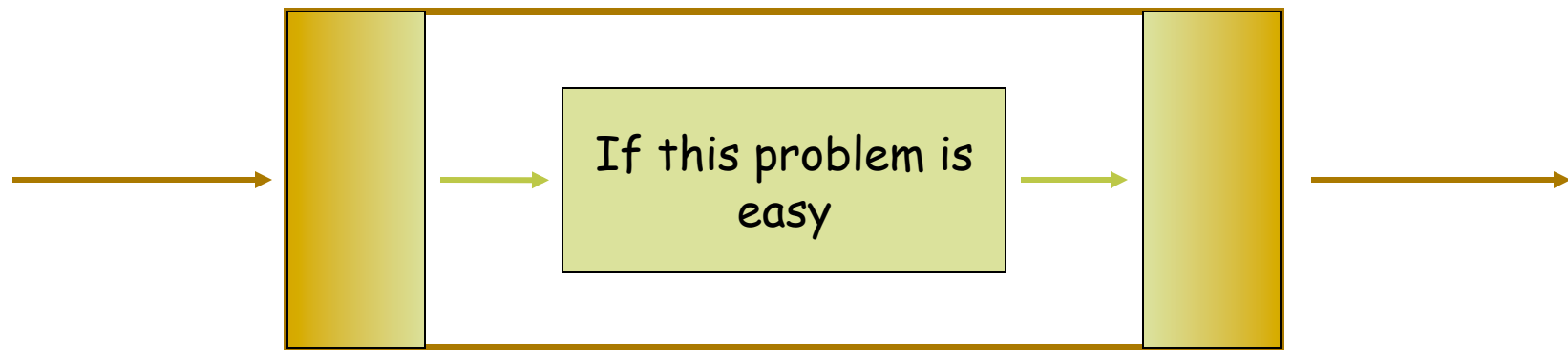
then so is this one.

*(as long as encoding/decoding is easy and compact)*



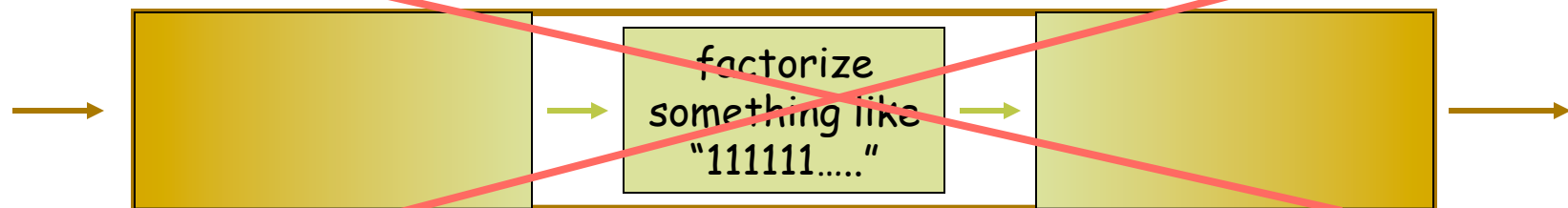
factorize something like "2007" **not as easy!**

# Reducibility



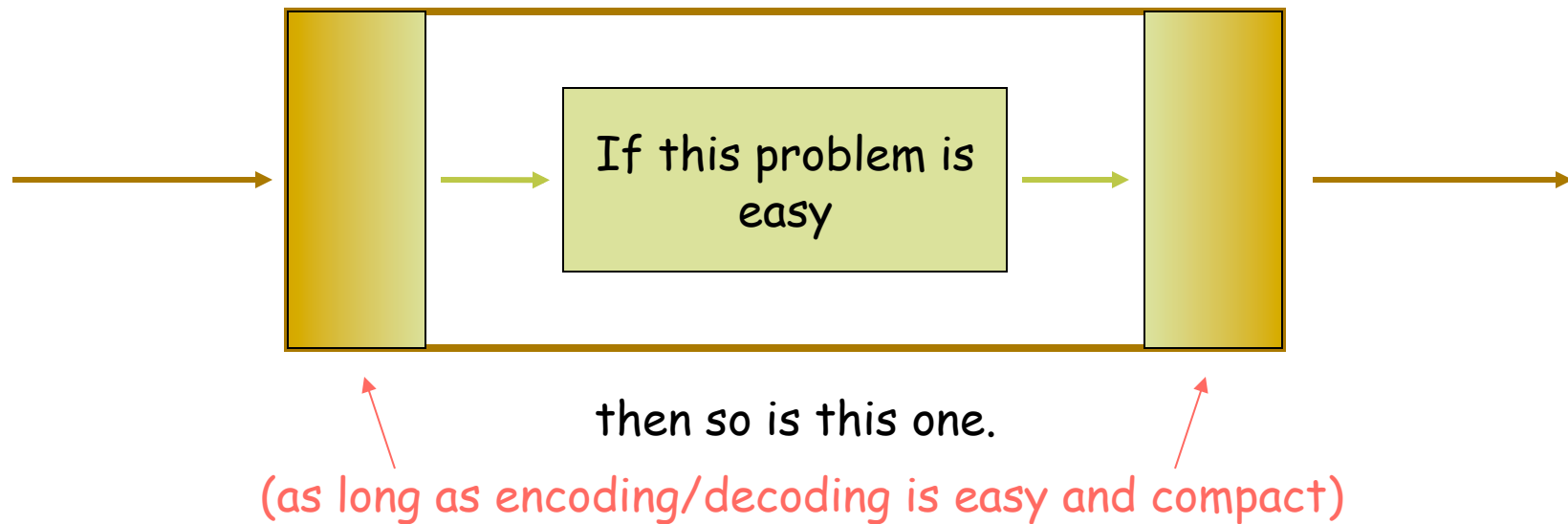
then so is this one.

*(as long as encoding/decoding is easy and compact)*



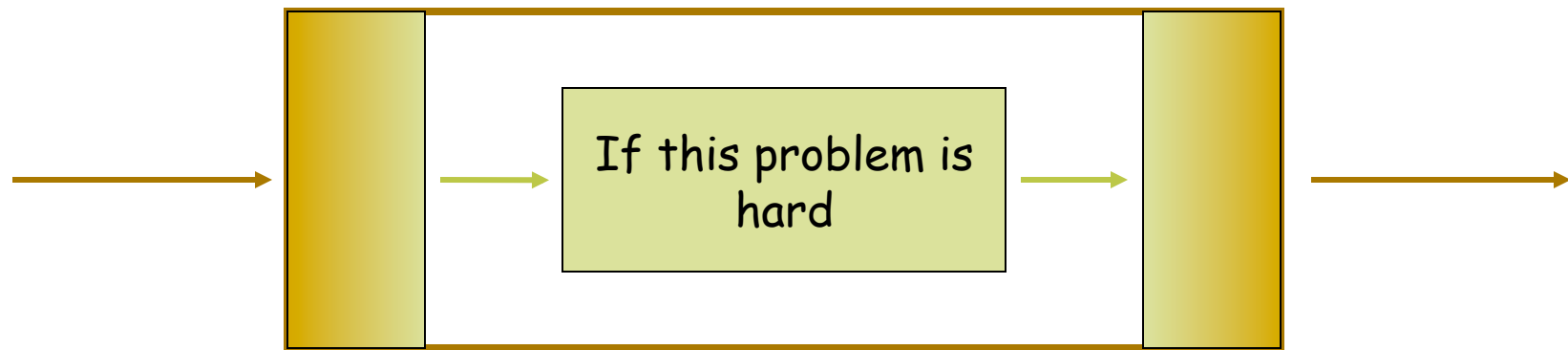
factorize something like "2007" *not as easy!*

# Reducibility



- What do we mean by a hard vs. an easy problem?
  - A hard problem has no fast solvers
  - An easy problem has at least one fast solver

# Reducibility

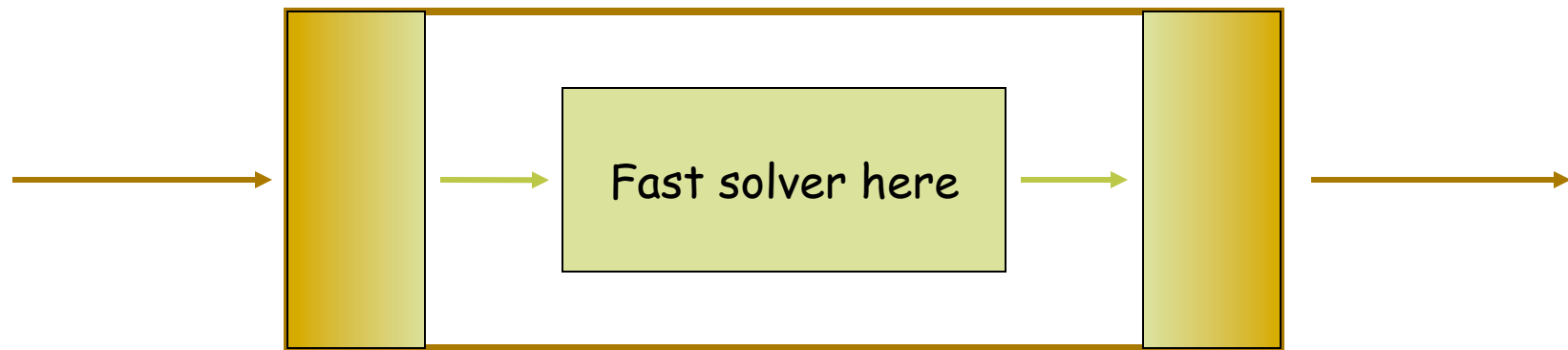


then is this one necessarily hard?

*(Nope. There might be a different, faster way to sort.)*

- What do we mean by a hard vs. an easy problem?
  - A hard problem has no fast solvers
  - An easy problem has at least one fast solver

# Reducibility

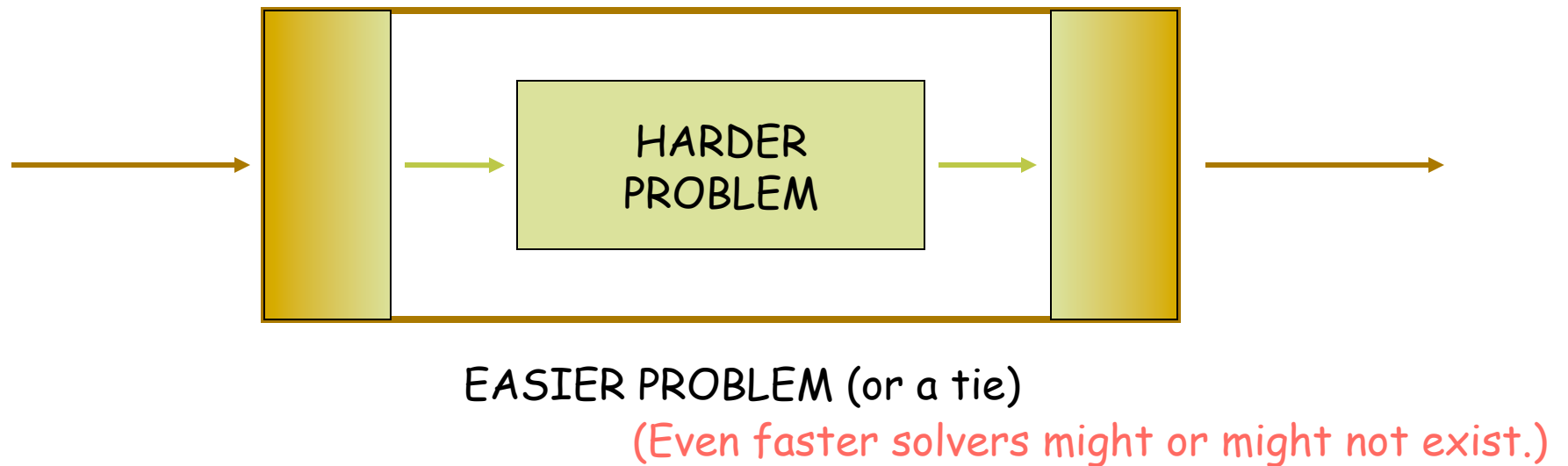


gives us a fast solver here.

*(Even faster ones might exist.)*

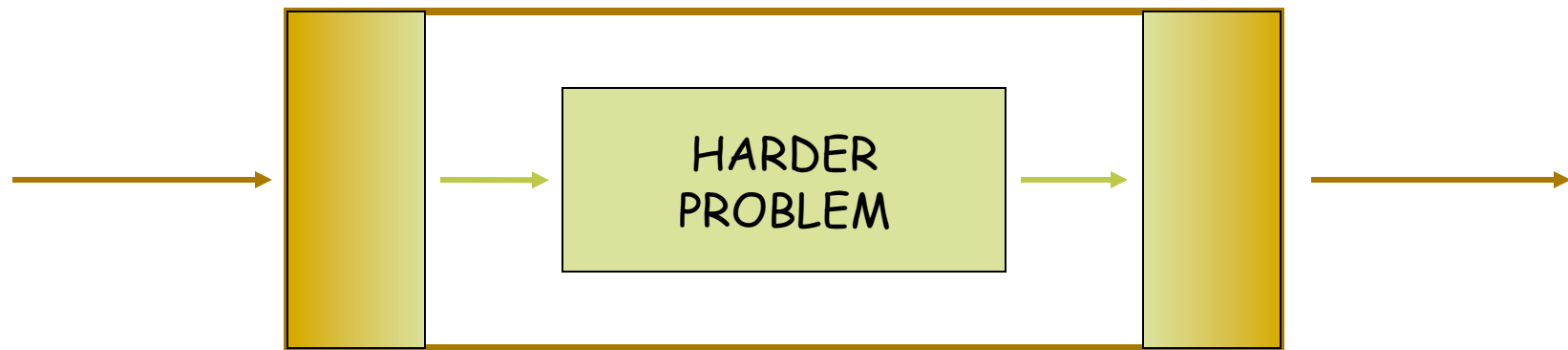
- What do we mean by a hard vs. an easy problem?
  - A hard problem has no fast solvers
  - An easy problem has at least one fast solver

# Reducibility



- What do we mean by a hard vs. an easy problem?
  - A hard problem has no fast solvers
  - An easy problem has at least one fast solver

# Interreducibility



EASIER PROBLEM (or a tie)

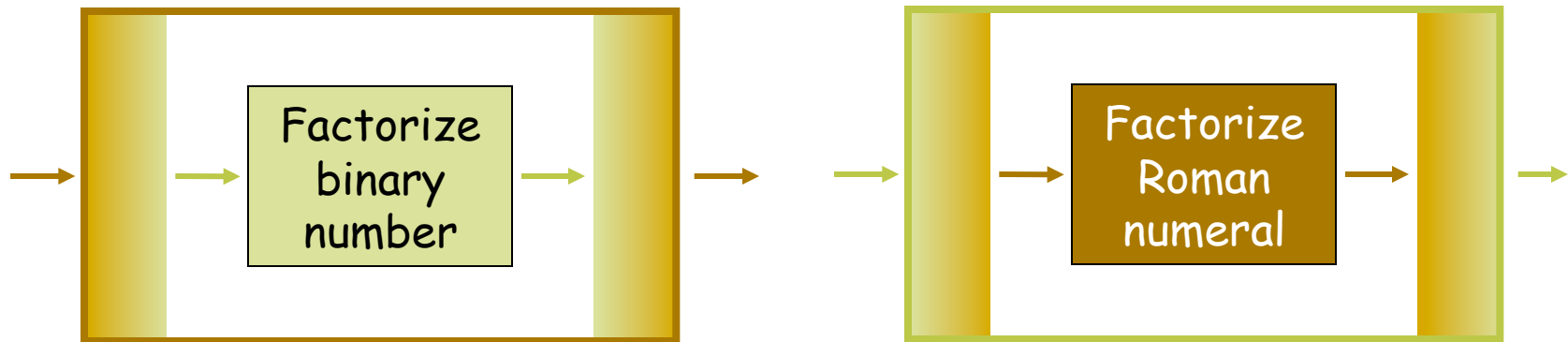
*(Even faster solvers might exist.)*

- What do we mean by a hard vs. an easy problem?
  - A hard problem has no fast solvers
  - An easy problem has at least one fast solver

# Interreducibility



EASIER PROBLEM (or a tie)

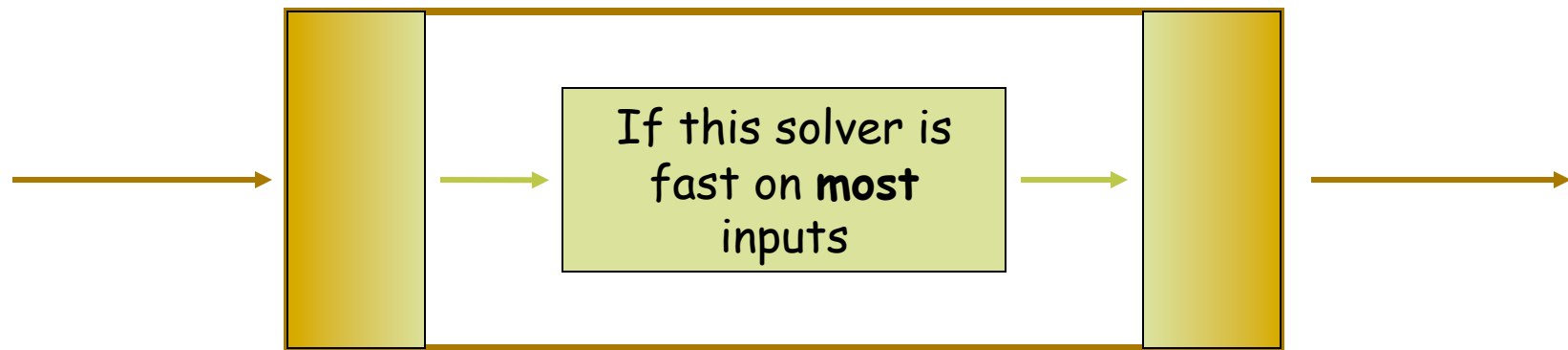


Factorize Roman numeral

Factorize binary number

**Interreducible problems**  
They must tie!  
Equally easy (or equally hard)  
if we're willing to ignore encoding/decoding cost.

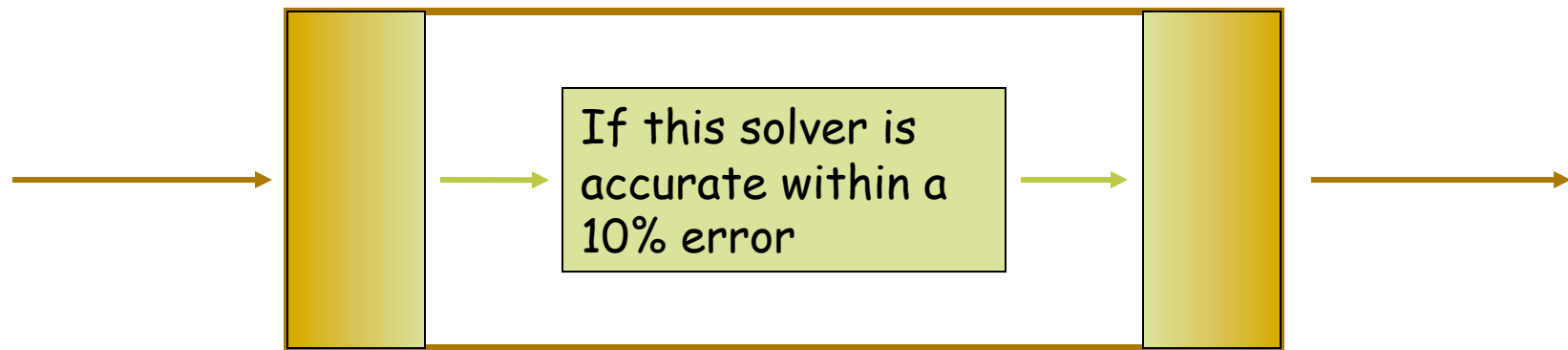
# Reducibility & “typical” behavior



then is this solver fast on **most** inputs?

Not necessarily - the encoding might tend to produce hard inputs for the inner solver

# Reducibility & approximation

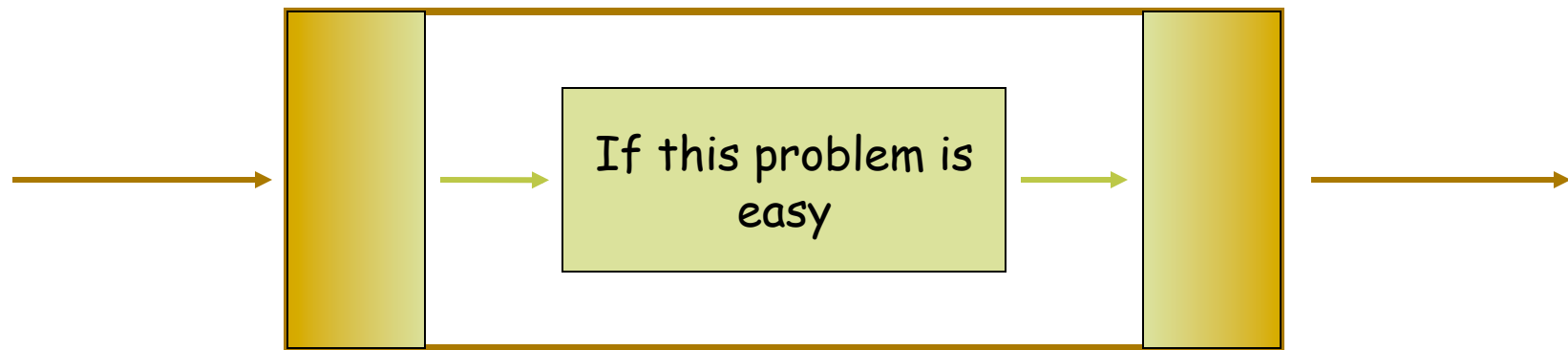


then is this solver equally accurate?

(Does almost-longest path lead to an almost-sort?)  
(Not the way I measure "almost"!)

- **Some** reductions (encode/decode funcs) may preserve your idea of "almost."
- In such cases, any accurate solver for the inner problem will yield one for the outer problem. Just be careful to check: don't assume reduction will work!

# Proving hardness

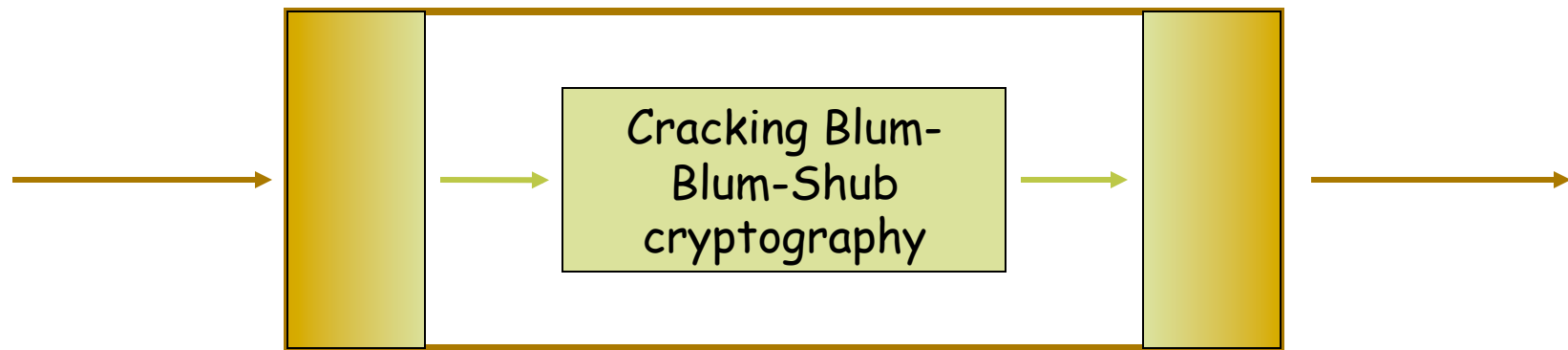


then so is this one.

So what can we conclude if outer problem is believed to be hard?

- What do we mean by a hard vs. an easy problem?
  - A hard problem has no fast solvers
  - An easy problem has at least one fast solver

# Proving hardness



Factoring integers

So what can we conclude if outer problem is believed to be hard?

- What do we mean by a hard vs. an easy problem?
  - A hard problem has no fast solvers
  - An easy problem has at least one fast solver