**Encodings and reducibility**

**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 $n$ = length of the input string

**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?

**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?

**Encoding integers**
- You want to write a solver that determines if an integer is prime
  - How hard is this? How hard is it to factor an integer?
- How do you encode the number 2010 as a string?
  - “2010” No harder: First convert from d digits to b bits in $O(d)$
  - “MMX” Then test primality in time $O(p) = O(4d^2) = O(4^d)$
  - Slightly harder for solver. Is decimal encoding isn’t harder; is it easier?
  - But easier for some users (ancient Romans)
  - “11111011010” If b = # bits, can factor in $O((\log b)^{b\log b})$.
    - Slightly easier for solver (if a PC)
  - But harder for most users
- Does it matter which of the above we use?
  - “2*3*5*67” (encode 2010 as its unique factorization)
    - Qualitatively different! Why?
    - Can test primality in $O(\text{length of input})$
  - “11111111111111111111111111111111…” (2010 times)
    - Qualitatively different! Why?
    - Can test primality in $O(\text{length of input})$

**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

Longest acyclic path in graph

Factorize Roman numeral

Factorize binary number

If this problem is easy

then so is this one.

(factorize something like "2007" not as easy)

(factorize something like "3*3*223" not as easy)

Reducibility

If this problem is easy

then so is this one.

(factorize something like "2007" not as easy)

(factorize something like "3*3*223" not as easy)

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

If this problem is hard
then is this one necessarily 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

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

Fast solver here gives us a fast solver here.

- 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

(Even faster ones might exist.)

Reducibility

EASIER PROBLEM (or a tie)
HARDER PROBLEM

- 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

Even faster solvers might or might not exist.

Interreducibility

HARDEST PROBLEM

EASIER PROBLEM (or a tie)

- 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

Even faster solvers might exist.

Interreducibility

Interreducible problems
- They must tie!
- Equally easy (or equally hard)
- If we’re willing to ignore encoding/decoding cost.

Factorize Roman numeral
Factorize binary numeral

Interreducible problems
then is this solver fast on most inputs?

- Not necessarily - the encoding might tend to produce hard inputs for the inner solver.
Reducibility & approximation

If this solver is accurate within a 10% error, 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 func) 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

If this problem is easy, then so is this one.

- 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

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

Proving hardness

Cracking Blum-Blum-Shub cryptography

What do we mean by a hard vs. an easy problem?