Recitation 1: Complexity review

6.5610, Spring 2023

February 10, 2023

1 Decision problems

- Informally, a *decision problem* (also called a *language*) is a set of rules that, to each input, assigns an output that is either 1 (a "yes instance") or 0 (a "no instance").
- Examples of decision problems:
  - Given a binary string, does it start with 3 zeros?
  - **Chess:** Given a chess board, can you force a win in the next move?
  - **Subset sum:** Given a set S of numbers, can you find a subset of them that sums to v?
  - **Hamiltonian path:** Given a graph G, does it contain a path that visits each node exactly once?
  - **Halting problem:** Given a program and an input, if you run the program on the input, will it terminate?
- More formally, a decision problem (or language) is a set $L \subseteq \{0,1\}^*$. The inputs to the problem are all possible bit strings in $\{0,1\}^*$. An input $x \in \{0,1\}^*$ is a "yes instance" if $x \in L$; otherwise, x is a "no instance."

2 Decidability

- Informally, a *program* is a constant-length string of code that implements an algorithm. On an input x, the program A produces some output $A(x)$. Note that the length of the program's code is independent of the size of the inputs. (Formally, we model programs as Turing machines.)
- A program *solves* a decision problem if it produces the correct output in $\{0,1\}$ on each input $x \in \{0,1\}^*$.
- A decision problem is *decidable* if it can be solved by a program that runs in a finite amount of time.

  Since the total number of programs (countably infinite) is much smaller than the total number of problems (uncountably infinite), not all problems are decidable. For example, the halting problem is undecidable.

3 Complexity classes and polynomial time (P)

- Within the set of decidable decision problems, we separate problems into multiple "complexity classes" based on the runtime of the programs that solve them, given inputs of length n:
  - **R**: the set of problems decidable in finite time.
  - **EXP**: the set of problems decidable in exponential time $2^{n^{O(1)}} = \exp(n)$.
  - **P**: the set of problems decidable in polynomial time $n^{O(1)} = \text{poly}(n)$. We say that these problems can be solved by "efficient algorithms."
- Time hierarchy theorems show that these classes are distinct: $\mathbf{P} \subsetneq \mathbf{EXP} \subsetneq \mathbf{R}$.

4 Non-deterministic polynomial time (NP)

- We just saw: **P** is the set of decision problems for which there exists a program A such that
  1. A solves the problem (i.e., on each input x, $A(x)$ is the correct answer to the problem), and
  2. for each input x of length n (i.e., $x \in \{0,1\}^n$), A runs in time poly(n) on input x.
- **NP** is the set of decision problems that are *verifiable* in polynomial time. More formally, **NP** is the set of decision problems for which there is a program V (called the "verifier") that takes in an input x of length n, along with a *witness* w of length poly(n), such that
  1. on any inputs, V runs in time poly(n),
  2. if x is a "yes instance," there exists some witness w such that $V(x, w)$ outputs 1.
  3. if x is a "no instance," there exists *no* witness w such that $V(x, w)$ outputs 1.
- We think of the witness as a "proof" that an input is a yes instance. Examples of witnesses:
  - **Chess:** The witness is the next move. The verifier checks whether this move forces a win.
  - **Subset sum:** The witness is a subset of the given numbers. The verifier checks whether these numbers sum to v.
  - **Hamiltonian path:** The witness is a path through the graph. The verifier checks whether (1) this is a valid path, and (2) the path visits each node in the graph exactly once.
- **P** $\subseteq$ **NP**: for any problem in **P**, the verifier could just ignore the witness and run the efficient algorithm A that solves the problem in polynomial time.
- **NP** $\subseteq$ **EXP**: for any problem in **NP**, an algorithm could solve the problem by enumerating every possible witness, and then running the verifier V on this witness.
- Open questions (which—if solved—would have a big impact on cryptography!): **P** = **NP**? **NP** = **EXP**?

5 Reductions

- Some problems in **NP** may be harder than others. We relate the complexity of various decision problems to each other using *reductions*.
- We say that problem A "reduces to" problem B (written as $A \leq B$) if a program that solves B can be used to solve A (so, intuitively, B must be at least as hard as A). More formally, a Karp reduction from A to B is a function f such that:

$$x \in A \implies f(x) \in B \qquad\qquad x \notin A \implies f(x) \notin B$$

- We say that A "polynomial-time reduces" to B if the reduction f from A to B can be computed in polynomial time (in the length of its input). In other words, A can be solved in polynomial time, plus whatever time it takes to solve B.
- A problem A is *NP-hard* if every problem in **NP** polynomial-time reduces to A. A problem is *NP-complete* if it is NP-hard and in **NP**.
- We think of all NP-complete problems as "equally hard." Finding a polynomial-time algorithm that solves an NP-complete problem would be a proof that **P** equals **NP**.

6 Cryptographic reducions

- In cryptography, we often prove that constructions are secure with a reduction. These reductions show that breaking a construction is "at least as hard as" breaking its assumptions.
- Examples of reductions:
  - We can prove that breaking a given encryption scheme is "at least as hard as" breaking a given PRG. To prove this, we show that if there is an (efficient) algorithm that breaks the encryption scheme, then there is an (efficient) algorithm that breaks the PRG.
  - We can prove that breaking a given PRG is "at least as hard as" factoring large integers. To prove this, we show that if there is an (efficient) algorithm that breaks the PRG, then there is an (efficient) algorithm that factors large integers.
- When working with efficient algorithms, we will use the concept of **negligible functions**:
  - Informally, a negligible function $f : \mathbb{N} \to \mathbb{R}$ is a function that grows slower than $\frac{1}{\text{poly}(n)}$.
  - Formally, for every polynomial p, there exists some x such that, for all $n > x$, we have that $f(n) < 1/p(n)$.
  - Useful property: an efficient algorithm should never observe an event that occurs with "negligible probability" (i.e., whose probability is determined by a negligible function).

Sources. These recitation notes are based off of the following handouts:
`https://61600.csail.mit.edu/2022/handouts/complexity-review.pdf`
`https://mit6875.github.io/FA22HANDOUTS/Complexity%20recitation%202022.pdf`