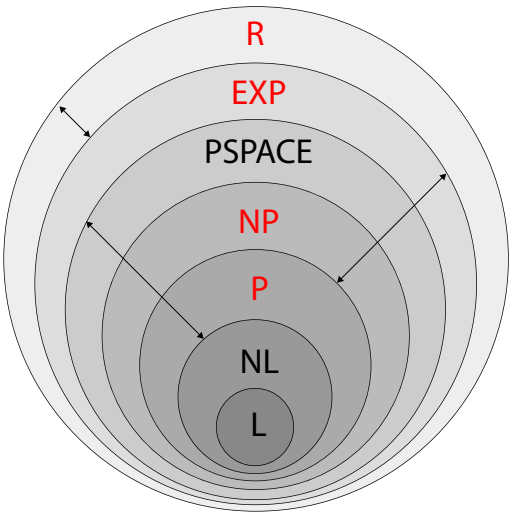
Lecture 23: Computational Complexity

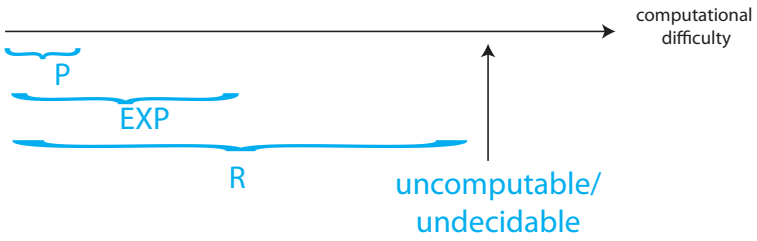
Lecture Overview

- P, EXP (EXPTIME), R
- Most problems are uncomputable
- NP
- Hardness & completeness
- Reductions



Definitions:

- P = {problems solvable in polynomial(n) (n^c) time} (Some people like to use $2^{O(\log n)}$)
  (what this class is all about – basically, anything that you can do "intelligently")
  P is strictly contained in EXP which is strictly contained in R.
- EXP = {problems solvable in exponential (2^{n^c}) time} (this is $2^{polynomial(n)}$)
- R = {problems solvable in finite time} "recursive" [Turing 1936; Church 1941]



Examples

- negative-weight cycle detection ∈ P
- $n \times n$ Chess ∈ EXP but ∉ P Who wins from given board configuration?
- Tetris ∈ EXP but don't know whether ∈ P Survive given pieces from given board.
- SAT: given a Boolean formula (and, or, not), is it ever true? x and not $x \rightarrow$ NO
  I.e., does there exist an assignment (x_1 = TRUE, x_2 = FALSE, etc.) of the variables in a given formula which results in the boolean outcome TRUE? P? EXP?

Halting Problem:

Given a computer program, does it ever halt (stop)?

- uncomputable ($\notin$ R): no algorithm solves it (correctly in finite time on all inputs)
- decision problem: answer is YES or NO

We won't actually do this, but one proof basically involves a counterexample program. Asssuming a halt-checking algorithm $h(M, x)$ exists, the counterexample program M_1 run on input x_1 is just "if $h(M_1, x_1)$ returns YES then loop forever, else halt"; i.e., it sends itself as input to the halting program, and does the opposite of whatever h says. You need to do a little bit more work to actually prove it properly, but that's the main idea.

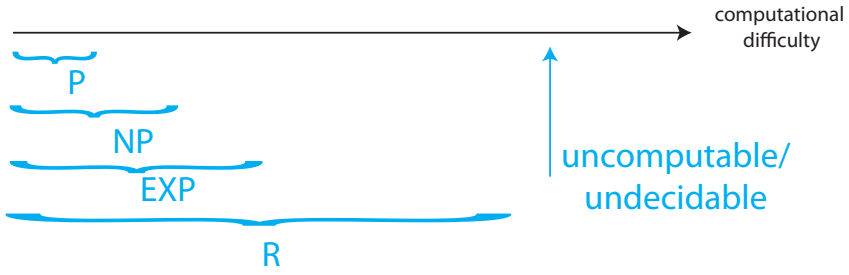
Most Decision Problems are Uncomputable

- program $\approx$ binary string $\approx$ nonneg. integer $\in \mathbb{N}$
- decision problem = function from binary strings ($\approx$ nonneg. integers) to {YES (1/TRUE), NO (0/FALSE)}. We can write this as a table, with each row being an input (binary string) leading to a boolean output; we can then take the list of boolean outputs and treat it as an infinite sequence of bits.
- $\approx$ infinite sequence of bits $\approx$ real number $\in \mathbb{R}$
  $|\mathbb{N}| \ll |\mathbb{R}|$: no assignment of unique nonneg. integers to real numbers ($\mathbb{R}$ uncountable)
- $\implies$ not nearly enough programs for all problems
- each program represents only one list of outputs, so there are an infinite number of output lists which are not represented by any program
- $\implies$ almost all problems cannot be solved

NP

NP stands for non-deterministic polynomial time
= {decision problems with solutions that can be "checked" in polynomial time}. This means that when the answer is YES, we can produce a "proof" and a polynomial-time algorithm can check this proof. (*E.g.*, a proof could be the set of assignments in a SAT instance.)
= {Decision problems solvable in polynomial time via a "lucky" algorithm}. The "lucky" algorithm can make lucky guesses, always "right" without trying all options.
Note: We won't cover why, but these two definitions are equivalent!

- nondeterministic model: algorithm makes guesses & then says YES or NO
- guesses guaranteed to lead to YES outcome if possible (no otherwise)



Example

SAT ∈ NP (as is Tetris)

- nondeterministic algorithm: guess TRUE/FALSE value of each variable
  (Tetris: guess each move, did I survive?)
- proof of YES: list of TRUE/FALSE assignments
  (Tetris: list what moves to make – rules of Tetris are easy)

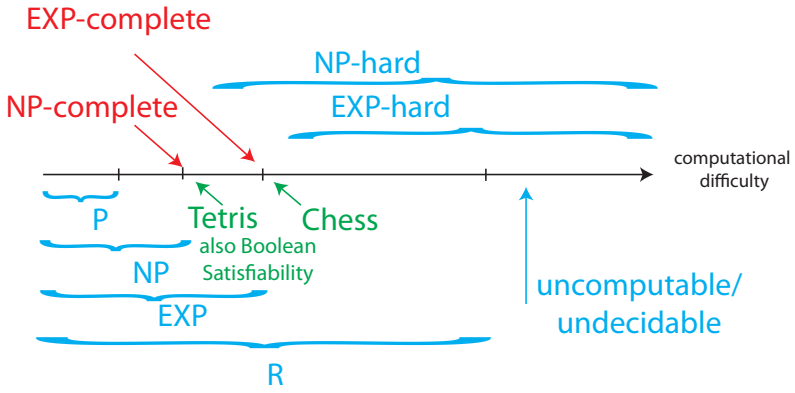
P ≠ NP

Big conjecture (worth $1,000,000). Some ways to summarize the hypothesis:

- finding [proofs of] solutions to problems can be harder than checking them (*e.g.*, checking a SAT assignment is easy)
- "Magic" is not a valid CPU branching instruction

Hardness and Completeness

SAT is NP-hard = "at least as hard as" every problem ∈ NP. In fact, SAT is NP-complete = NP ∩ NP-hard.



Similarly

Chess is EXP-complete = EXP ∩ EXP-hard. EXP-hard is as hard as every problem in EXP. If NP ≠ EXP, then Chess ∈ EXP \ NP. Whether NP ≠ EXP is also an open problem but less famous/"important". If there's a polynomial bound on the number of moves, then it is PSPACE-complete.

Reductions

Convert your problem into a problem you already know how to solve (instead of solving from scratch – we've been doing a lot of this with graph problems recently!)

- most common algorithm design technique (in reverse): use a "subroutine". As long as the rest of your program – the process of converting your input into a form suitable for the subroutine, and the process of converting the output from the subroutine into output suitable for your program – is polynomial-time, you're ok!
- unweighted shortest path → weighted (set weights = 1)
- Computer scientist trying to boil water

All the above are One-call reductions: A problem → B problem → B solution → A solution
Multicall reductions: solve A using free calls to B — in this sense, every algorithm reduces problem → model of computation
NP-complete problems are all interreducible using polynomial-time reductions (same difficulty). This implies that we can use reductions to prove NP-hardness — such as in 3-Partition → Tetris. So to prove that X is NP-hard, just show that you can use a (hypothetical) subroutine that solves X in a program that solves SAT, such that the rest of the code (other than the hypothetical subroutine) runs in polynomial time.

Examples of NP-Complete Problems

- 3-Partition: given n integers, can you divide them into triples of equal sum?
- Traveling Salesman Problem: shortest path that visits all vertices of a given graph — decision version: is minimum weight $\leq x$?
- longest common subsequence of k strings
- Minesweeper, Sudoku, and most puzzles
- shortest paths amidst obstacles in 3D
- find largest clique in a given graph
- subset sum: given a set of integers, does any subset sum to 0 (or a given integer s)