COMPUTATIONAL COMPLEXITY

CSC 300

INTRODUCTION

- At the beginning of the semester, we started discussing computational complexity with Big-O notation.
- We placed programs into different categories (or sets) based on their runtime with generalized input n.
  - Linear Time: $O(n)$.
  - Logarithmic Time: $O(\log(n))$
  - Polynomial Time: $O(n^c)$.
  - Exponential Time: $O(2^{n^c})$.
  - And products of the above.

INTRODUCTION CONTINUED

- We will now take a more broad look at sets of programs in regards to not just their complexity time, but their solvability.
- The following material could be taught over an entire course, but we are going to condense it into a single lecture. *thumbs up*
- Topics include:
  - P, EXP, R
  - The fact that most problems are incomputable.
  - NP
  - Hardness and Completeness
  - Reductions

DEFINITIONS

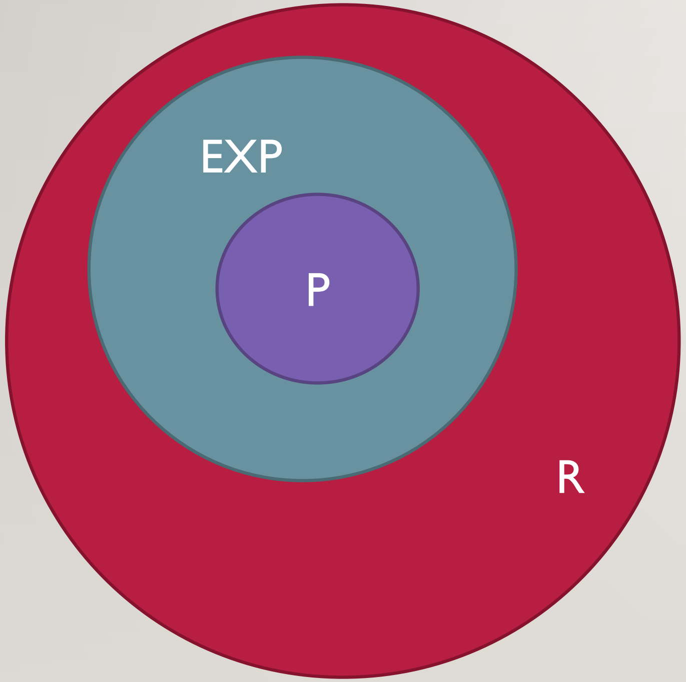
- Problems: To be more precise, they are referred to as **decision problems**.
  - Decision problems have a **yes** or **no** answer.
- P = {Problems solvable in polynomial time, $O(n^c)$.}
  - P stands for *polynomial*.
  - Notice the curly braces. This indicates that P is a set of all the problems solvable in polynomial time.
  - This includes all of the programs and algorithms that you've done so far.
- EXP = {Problems solvable in exponential time, $O(2^{n^c})$.
  - EXP stands for exponential.
  - The programs will still finish (in theory), although the time may not be reasonable.

DEFINITIONS CONTINUED

- R = {Problems solvable in finite time.}
  - R stands for recursive.
    - When this theory was being developed in the 1930's, recursive had a different meaning; which was *finite*.
  - Again, the programs will finish in theory, but the time may be up to *near* infinite.
- A program that is not in the set R is considered incomputable.

COMPUTATIONAL DIFFICULTY GRAPH

- $P \subseteq EXP \subseteq R$



WHAT IS REASONABLE TIME?

- **Cobham-Edmond's Thesis** asserts that computational problems can be feasibly computed in some computational device only if they can be computed in polynomial time; this is, they fall in set P.
- Can be found in Alan Cobham's 1965 paper "The intrinsic computational difficulty of functions".
- Simply states that problems in P are "easy, fast, and practical", while problems not in P are "hard, slow, and impractical."
- This is a very generalized view; some computer scientists disagree with the vagueness and the fact that Cobham ignores:
  - Constant factors and lower-order terms.
  - The size of the exponent.
  - The typical size of the input. (Who cares of an algorithm is 2^n if n is always less than 5?)
- So time complexity is "relative" to the input and does not have the same explicit meaning as standard time.

EXAMPLES OF PROBLEMS

- Any of the algorithms we're used so far in the class are elements in the set P.
  - {Searching, sorting, shortest path, etc.} ∈ P
- $n \times n$ chess ∈ EXP, but ∉ P.
  - Putting chess in terms of a decision problem, it would be: Given a board configuration, does white/black win?
  - Looks at all possible strategies. 8x8 is very difficult.
  - The game Go, and lots of other games fit into this category.
- Tetris ∈ EXP. Unknown if it can be solved in polynomial time.
  - Given a board, can we survive given a sequence of pieces?

EXAMPLE OF AN INCOMPUTABLE PROBLEM

- The **Halting Problem** asks: Given a computer program, does it ever halt (stop)?
  - Issues include, infinite loops, bugs; does the program just go on forever?
  - Not in the set R.
- While some programs are solvable, there is no algorithm to determine this for *all* programs in finite time.
  - There is a proof for this, but it's outside the focus of this class.

THE MAJORITY OF PROGRAMS ARE INCOMPUTABLE

- The far majority of *decision problems* $\notin$ R.
- Recall that decision problems have yes/no (binary) answers.
- Think about how to generalize all computer programs.
  - Technically, all programs are reduced to a finite binary string.
  - This binary string represents an integer, that is to say a natural number $\in \mathbb{N}$.
- Now think about the space of all decision problems.
  - A function that maps input to a yes/no output.
  - The input can be considered a binary string $\in \mathbb{N}$, or integer.
  - Output is $\{0,1\}$. That is, zero or one.

THE SET OF PROGRAMS ∉ R

- Now, we could represent these functions with specific input/output as a table. Below is hypothetical.

Input	0	1	2	3	4	5	6	7	…
Output	0	0	1	0	1	1	1	1	…

- We then have an infinite number of bits representing the output. This represents our decision problem.
- A program is represented by a <u>finite</u> string of bits.
- A decision problem is represented by an <u>infinite</u> string of bits.
- Let's get theoretical and say we place a decimal point (.) in front of the string of 0/1 bits. Then the string represents a real number between [0,1].
- Therefore, any real number between [0,1] can be represented with an infinite number of bits available.
- So decision problems are in the set of all *real* numbers.

THE SET OF PROGRAMS VS THE SET OF DECISION PROBLEMS

- So, we have that all programs $\in \mathbb{N}$ and all decision problems $\in \mathbb{R}$.
- $|\mathbb{R}| \gg |\mathbb{N}|$.
- The set of real numbers $|\mathbb{R}|$ is uncountably infinite, while the set of natural numbers $|\mathbb{N}|$ is countably infinite.
- Bad news bears. That means there are way more problems than we have programs to solve them.
- Technically, almost every problem is unsolvable by any program.

DEFINITION OF NP

- NP = {Decision problems solvable in polynomial time via a "lucky" algorithm.}
  - Here, "lucky" means that when choices are presented, the program *always* chooses the correct choice *without* trying all options.
  - NP stands for Nondeterministic Polynomial model: The algorithm makes guesses, and then says YES or NO.
    - A deterministic program has only one path to follow given a state.
    - A non-deterministic program may allow for different paths for any given state. (Choices)
  - Guesses guaranteed to lead to YES outcome if possible, (otherwise NO).
  - Not realistic, can't be built on any real computer. But theory is useful.

NP CONTINUED

- Another (more useful) definition for NP is:
- NP = {decision problems with solutions that can be "checked" in polynomial time.}
  - When the answer is YES, we can prove it in polynomial time by checking the choices made and verifying that YES is the given output.
  - Easier to check if solutions are correct than to generate proofs of solutions.

EXAMPLE OF NP

- Tetris $\in$ NP. The nondeterministic algorithm guesses each move. Did I survive?
- Proof is fairly easy. Given a board and series of pieces and a YES solution, go back and use the inputs for left/right and rotation, and verify the answer.
  - The rules of Tetris are fairly easy.

IS P $\neq$ NP?

- It is unknown if P $\neq$ NP. That is, are all decision problems that can be checked in polynomial time able to be solved outright in polynomial time?
- It is a fairly large conjecture to assume this is the case.
- Proving the above is worth $1,000,000. Would also make you the most famous computer scientist of our era.

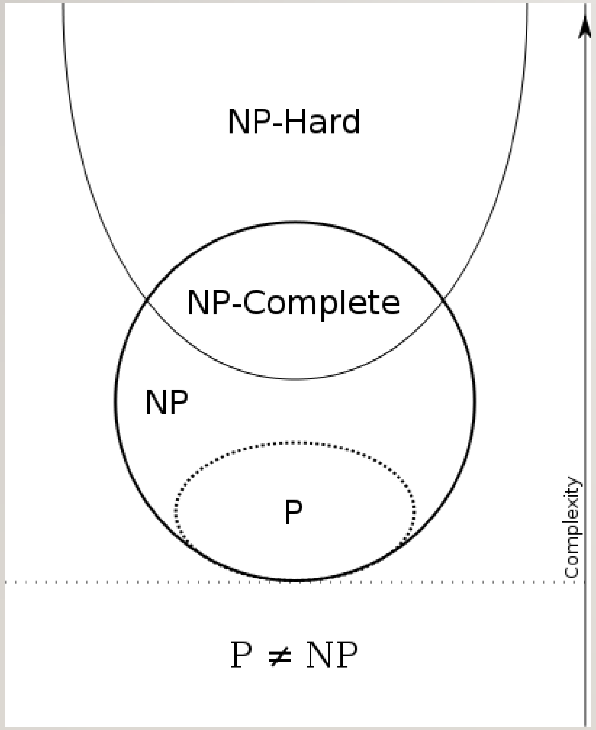
NP-HARDNESS

- NP-hard = {Decision problems that are *at least* as hard as the every other problem in NP.}
- It is suspected that there are no polynomial time algorithms for NP-hard problems.
  - Never been proven.
- If P $\neq$ NP (assumed), then NP-hard problems cannot be solved in polynomial time.
- A common example is the travelling salesman problem:
  - Find the least-cost cyclic route through all nodes of a weighted graph.
- In many cases, *heuristics* $\in$ P are used in place of exact solutions that are found to be NP or more difficult.

NP-COMPLETE

- NP-Complete = {Decision problems that are both NP and NP-Hard).
  - Shown as NP ∩ NP-hard.
- We'll get to reduction soon, but the main difference between NP-complete and NP is that an NP-complete decision problem, *A*, is one where all known NP problems can be reduced to *A*.
  - For now, think of reduction as simplifying a decision problem such that given the same inputs, the exact same output is given for both problems.

GRAPH OF NP-HARDNESS: ASSUMING P ≠ NP



EXAMPLES OF NP-HARD

- Tetris is actually NP-hard.
  - It is "as hard as" every problem in $\in$ NP.
  - In fact, Tetris is **NP-complete**. Because it is NP *and* NP-hard.
- Similarly, chess is EXP-complete.
  - Chess is both EXP and EXP-hard.
- Currently unknown if NP $\neq$ EXP.
  - Not as famous of a problem, but still important.

REDUCTION

- A **Reduction** is when we convert a problem into a problem we already know how to solve.
  - Sometimes easier than solving the problem from scratch.
- This happens to be one of the most common algorithm design techniques used by theorists.

REDUCTION EXAMPLES

- How do we solve the shortest path problem algorithmically when the graph is unweighted?
- Min-product path. How do we find the path with the smallest product of it’s weighted edges?
- How do we find the longest path of a weighted graph?

REDUCTION EXAMPLES

- How do we solve the shortest path problem algorithmically when the graph is unweighted?
  - Set all weights = 1.
- Min-product path. How do we find the path that minimizes product of it's weighted edges?
  - Take logs. (Converts products to sums, then equivalent to shortest path).
- How do we find the longest path of a weighted graph?
  - Negate weights and run same algorithm.

REDUCTION CONTINUED

- All of the previous examples are referred to as **One-call reductions**:
  - A problem - > B problem -> B solution -> A solution.
  - Simple, powerful, and very useful.
- **Multi-call Reductions**: Solve A using free calls to B.
  - In this sense, every algorithm reduces the problem in this model of computation.

REDUCTIONS AND NP-COMPLETE

- By definition, all NP-Complete programs can be reduced to each other.
- Can prove NP-hardness of a given algorithm by attempting to reduce problem to another NP-Complete problem.
- For example, the 3-partition problem can be reduced to Tetris, that is:
  - 3-partition problem -> Tetris.
  - The 3-partition problem is that given n numbers, can I divide them into three groups where the sum of all the groups are equal. (Proven by Karp to be NP-complete).
  - So focus on showing that Tetris is *at least as hard as* 3-partition.

MORE NP-COMPLETE PROBLEMS

- Travelling Salesman Problem
- Longest common subsequence for n strings.
- Minesweeper, Sudoku, most puzzle games.
- Shortest path in a 3D setting.
- SAT: aka the Boolean Satisfiability problem. Given a set of variables comprising a Boolean formula, is the answer TRUE or FALSE?
  - Example: (x AND y) OR NOT z
  - This was the first problem to be proven to be NP-Complete. Next group of NP-Complete algorithms were reduced to SAT to prove that they were NP-Complete.