CCS Problem-Solving II | Professor: Padraic Bartlett

Project 4: NP-Completeness

Weeks 1-10 | *UCSB 2015*

An **algorithm** is a precise and unambiguous set of instructions.

Typically, people think of algorithms as a set of instructions for solving some problem; when they do so, they typically have some restrictions in mind (namely, "simple" and "easy to perform") for the kinds of instructions they consider to be valid. For example, most people think that the following process for proving the Riemann hypothesis is not really an algorithm:

1. Prove the Riemann hypothesis.
2. Rejoice!

Conversely, people do regard the following algorithm for multiplying two numbers using only addition as a valid algorithm:

Algorithm. Take as input any two positive integers $x < y$. Consider the following algorithm:

0. Define a new number *prod*, and initialize it (i.e. set it equal) to 0.
1. If $x = 0$, stop, and return the number *prod*.
2. Otherwise, set $prod = prod + y$ and $x = x - 1$.
3. Go to step 1.

Another way to multiply numbers with only "basic" operations (i.e. addition, division by two round down) is the process of "peasant multiplication:"

Algorithm. Take as input any two positive integers $x < y$. Consider the following algorithm:

1. Define a new number *prod*, and initialize it (i.e. set it equal) to 0.
2. If $x = 0$, stop, and return the number *prod*.
3. Otherwise, if x is odd, set $prod = prod + y$.
4. Regardless of what x was in the step above, set $x = \lfloor x/2 \rfloor$, and $y = y + y$.
5. Go to step 2.

Roughly speaking, this algorithm succeeds because we can write

$$x \cdot y = \begin{cases} 0, & x = 0, \\ \lfloor x/2 \rfloor \cdot (y + y), & x \text{ even}, \\ \lfloor x/2 \rfloor \cdot (y + y) + y, & x \text{ odd}, \end{cases}$$

and this algorithm is a repeated application of this fact.

One interesting thing about peasant multiplication is how much faster it is as compared to our first multiplication algorithm! For example, peasant multiplication takes just 7 runs to take the product of $123, 231$ (try this!) Conversely, our first algorithm for multiplying would have taken us 231 runs.

This above distinction motivates us to think about the idea of **runtime**. In general, given inputs x, y, you can see that it will take x loops to multiply two numbers with our first algorithm (because x decreases by 1 at each step), and that it will take at most $\log_2(x)$ loops for our peasant multiplication algorithm to complete (because x is replaced by $\lfloor x/2 \rfloor$ at each step.)

Based on this, we say that peasant multiplication has **runtime**[1] $O(\log_2(\min(m, n)))$, on inputs n, m based on this analysis: roughly speaking, the worst peasant multiplication can do is take $\log_2(\min(m, n))$ runs to complete, and therefore needs at most some constant times $\log_2(\min(m, n))$ many second to multiply its numbers, if each of the operations we're using is "easy" (i.e. can be performed in some constant amount of time.)

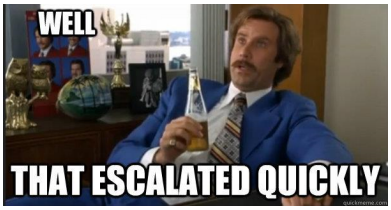
Notice that this compares favorably to the normal multiplication algorithm we gave before, which needs at least $\min(m, n)$ loops to multiply two numbers, and is thus something that we think has runtime $O(\min(m, n))$.

So: we have two algorithms to solve the same problem, one of which has a demonstrably shorter runtime than the other! This raises a natural question: suppose we have an algorithm that purports to solve a problem. When can you find a faster algorithm? When **should** you try to find a faster algorithm?

The following table might help illustrate things some. Below, we plot five functions with runtimes $n, n^2, n^3, n^5, 2^n$ versus input sizes for n ranging from 10 to 50, with the assumption that we can perform one step every 10^{-6} seconds.

Runtime v. Input	10	20	30	40	50
n	$1 \cdot 10^{-5}$ sec.	$2 \cdot 10^{-5}$ sec.	$3 \cdot 10^{-5}$ sec.	$4 \cdot 10^{-5}$ sec.	$5 \cdot 10^{-5}$ sec.
n^2	$1 \cdot 10^{-4}$ sec.	$4 \cdot 10^{-4}$ sec.	$9 \cdot 10^{-4}$ sec.	$1.6 \cdot 10^{-3}$ sec.	$2.5 \cdot 10^{-3}$ sec.
n^3	$1 \cdot 10^{-3}$ sec.	$8 \cdot 10^{-3}$ sec.	.027 sec.	.064 sec.	.125 sec.
n^5	.1 sec.	3.2 sec.	24.3 sec.	1.7 min.	5.2 min.
2^n	$1 \cdot 10^{-3}$ sec.	1 sec.	17.9 min.	12.7 days.	35.7 years

In this table, the functions n, n^2, n^3, n^5 all grow at roughly not-awful rates. For 2^n, though ...



The issue with an algorithm that runs in exponential time, like 2^n, is that very small changes in the size of the input lists can create massive increases in the time needed to run the program. For example, in the list above, an increase in the number of elements

[1] Formally: we say that some positive function $f(n)$ is $O(\log_2(\min(m, n))$ if there is some constant M such that $\frac{f(n)}{n^2} \leq M$. Roughly speaking, this says that $f(n)$ grows "like" $\log_2(\min(m, n)$, up to some fixed constant multiplier. This idea extends to $O(n), O(n^2)$, and really $O(anything)$.

examined by 40 (a potentially very paltry increase, if you're like sorting lists with thousands of elements) increased our runtime from .001 seconds to 35.7 **years**.

So: polynomials good, exponentials bad. To make this rigorous, here are a few definitions:

Definition. A problem is said to be of class P if there is an integer k and algorithm A that solves instances of length n of this problem with runtime $O(n^k)$.

Definition. A problem R is said to be of class NP if it has "efficiently verifiable proofs." Specifically, we ask for the following two things:

- Given any instance I for which our problem is "true," there is a proof of this claim.
- There is an algorithm A that, given any proof that claims an instance I of our problem is true, can verify whether this proof actually corresponds to I being true in polynomial time.

Roughly speaking, P is the set of problems we can quickly "solve," while NP is the set of problems that we can "quickly check solutions for." Notice that any problem in P is in NP, because to quickly check any solution we could just solve the problem (as it's in P!)

A very strange thing that happens with NP is the following: we say that a problem Q is **NP-complete** if it is in NP, and also that any algorithm that solves Q can be used to solve any other problem in NP with an at most polynomial increase in its runtime. Roughly speaking, a NP-complete problem is a sort of "universal" problem for NP — if you can solve a NP-complete problem quickly (i.e. in polynomial time,) then you can solve **every** problem in NP quickly!

The surprising thing with NP is the following:

- There are NP-complete problems.
- Almost every problem[2] in NP that is not known to be in P seems to be NP-complete.

This leads to a particularly fun and silly branch of open problems:

1. Take any popular sort of puzzle (like Sudoku, or Minesweeper.)
2. Find a way to generalize that puzzle to inputs of size n, instead of whatever form the puzzle traditionally comes in.
3. Show that this puzzle is a NP-complete task. (See here for the "Minesweeper is NP-complete" proof, and here for some notes on NP-completeness and Latin squares, which you can extend with a little work to Sudoku grids.)

In this project, students will study and learn about P versus NP, pick out a class of puzzles of their choosing, and attempt to determine which are NP-complete!

[2]It is open whether P = NP, and also whether every problem in NP not in P is NP-complete. You will probably not solve these problems in this project.