Math 7H | Professor: Padraic Bartlett

Lecture 8: Algorithms

Week 8 | *UCSB*

These notes are meant to serve as a quick introduction to the concept of an **algorithm**. In this talk, we will go over the basics that we'll use in future talks on P vs. NP and other research questions at the intersection of theoretical computer science and mathematics.

If you're interested in a longer and (I believe) more elegant treatment, Jeff Erickson's online course notes on algorithms are some of the best-written notes I've read on the subject! Check them out.

1 Defining Algorithms

Definition. 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 for the kinds of instructions they consider to be valid. For example, consider the following algorithm for proving the Riemann hypothesis:

1. Prove the Riemann hypothesis.
2. Rejoice!

This is not a terribly useful algorithm. Typically, we'll want to limit the steps in our algorithms to mechanically-reproducible steps: i.e. operations that a computer could easily perform, or operations that a person could do with relatively minimal training.

For example, suppose that your space of allowable steps were operations you could perform with a compass and straightedge: specifically, suppose that you can connect points with a straight line using the straightedge, and draw a circle with a given center through any other point with the compass. Then you can solve the following problem:

Problem. Take any line L in $\mathbb{R}^2$ and any point P not on L. Can you construct a line through P that's perpendicular to L?

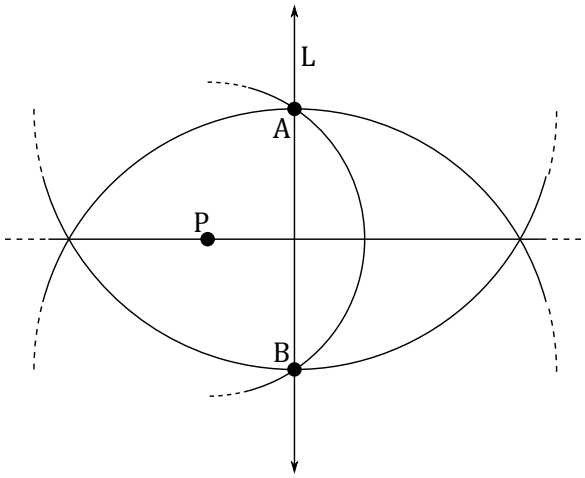
Algorithm. Take as input any line L and point P not on L, and consider the following process:

1. Pick any point on L. Call it A.
2. Draw a circle centered at P through A.
3. If this circle only intersects L at A and no other point, draw a line through A and P. This line is perpendicular to L.
4. Otherwise, it intersects L at some other point B. Draw a circle through A centered at B, and another circle through B centered at A.

5. These two circles must intersect at two points: find them and call them C, D.

6. Connect C and D. This is a line that is perpendicular to L and that goes through P.

We illustrate a sample run of our algorithm below:



Note that algorithms do not need to contain within themselves a proof that they work; while the above process indeed creates a perpendicular line to L through P, it does not explain the geometry behind why it does so. In general, when we construct algorithms, it's good form to possibly explain after we've done so **why** our algorithm performs the task we claim it does; while we won't always do this in this class, we will attempt to do so whenever we study an algorithm whose functionality isn't immediately obvious. Also, note that we've included a sample run of our algorithm after describing it. This is one of the most important things you can do with any algorithm, whenever possible; it can often turn a difficult and confusing writeup into a much clearer solution. Do this!

A second problem you might want an algorithm to solve is the task of "multiplication." On one hand, you might think that this is a trivial problem to write an algorithm to solve:

Algorithm. Take as input any two positive integers n, m. Consider the following algorithm:

1. Calculate $n \cdot m$. Return this number.

This is kind-of silly, but is not actually an incorrect answer! In many situations, we may want to think about multiplying two numbers together as a "simple" task, and thus something we can do in one step! As with many situations, what a valid algorithm considers as its possible "steps" will be something we need to define for each problem!

So, let's try this again, with some interesting constraints; suppose your only simple operations are addition and subtraction. Can you multiply two numbers together?

The answer is still yes, by using the idea that multiplication is repeated addition:

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.

We illustrate this algorithm below, on input $x = 4, y = 6$. The rows of the table below correspond to the values of our variables after each complete loop of the above instructions:

run	*prod*	x	y
initialization	0	4	6
1	6	3	6
2	12	2	6
3	18	1	6
4	24	0	6
stop	24	–	–

Another way to multiply numbers with only 'basic" operations is the process of "peasant multiplication," a process for multiplying large numbers known since the seventeenth century BC that allows relatively innumerate people to perform otherwise-difficult calculations.

Problem. Take any two positive integral numbers x, y, and suppose that the only operations you have are the ability to add two numbers, divide a number by two (rounding down), and to calculate the parity (even or odd) of any number. Can you efficiently calculate $x \cdot y$?

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.

We illustrate this algorithm below, on input $x = 123, y = 231$:

run	*prod*	x	y
initialization	0	123	231
1	231	61	462
2	693	30	924
3	693	15	1848
4	2541	7	3696
5	6237	3	7392
6	13629	1	14784
7	28413	0	29568
stop	28413	–	–

One interesting thing about the above process is how much faster it ran than our first process! That is; this took us just 7 runs to take the product of $123, 231$; conversely, our first algorithm for multiplying would have taken us 231 runs!

2 Algorithms and Runtime

This above distinction motivates us to think about the idea of **runtime**. For example, when we put the pair $123, 231$ into the peasant multiplication algorithm above, it took us 7 runs to find their product! In general, suppose that we want to find the product of two numbers x, y. Each pass of our process replaces x with $\lfloor x/2 \rfloor$, and our process ends when $x = 0$. So, for example,

- if $x = 1$, this takes one pass;
- if $x = 2$, this takes 2 passes ($2 \to 1 \to 0$),
- if $x = 4$, this takes 3 passes ($4 \to 2 \to 1 \to 0$),
- if $x = 8$, this takes 4 passes ($8 \to 4 \to 2 \to 1 \to 0$),
- if $x = 16$, this takes 5 passes ($16 \to 8 \to 4 \to 2 \to 1 \to 0$),
- ...
- and if $x = 2^n$, this takes $n + 1$ passes ($2^n \to 2^{n-1} \to \ldots 4 \to 2 \to 1 \to 0$).

These cases all represent the "worst-case" scenario for our algorithm, in that $\lfloor x/2 \rfloor$ decreases even faster when we plug in an odd number. So, in general, we say that our process needs $\log_2(x)$ loops to complete, given inputs (x, y). Also, notice that if we wanted to make this algorithm faster, we could switch x, y at the start of our process so that x is always the smaller of the two numbers (we can do this because $x \cdot y = y \cdot x$!)

We say that peasant multiplication has runtime $O(\log_2(\min(m, n)))$, on inputs n, m based on this analysis: roughly speaking, the worst peasant multiplication can do is take $\log_2(\min(m, n))$ passes, and therefore needs at most some constant $\cdot \log_2(\min(m, n))$ many steps to multiply its numbers. (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 $O(anything)$.)

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?

3 Polynomials and Exponentials

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. In other words, if you have an algorithm that runs in time n^3, and you want to increase the size of n you work with by one, say, you're increasing the number of steps by

$$(n+1)^3 - n^3 = 3n^2 + 3n + 1$$

many steps, which is much smaller than n^3 for any decently large n. In other words, this intuitively says that if your algorithm can run in reasonable time and find a solution for a set of some size n, it can find a solution for a set of size $n+1$ in not that much more time. Indeed, if we look at the table, this is what we see: even for n^5, if we could run our problem for a set of size 10 we could probably still run it for size 50 if we hunted down some nicer hardware.

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 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**. It doesn't matter if we find faster computers, or if we simply get **all** the computers; if our algorithm has $O(2^n)$ runtime, we're going to have a hell of a time running it for all but a few values of n.

4 P versus NP: Definitions

So: polynomials good, exponentials bad.

To make this rigorous, here are a few definitions:

Definition. A **yes-no problem** is some well-defined task that we want to solve in general, that has an answer that is either true or false. Examples are things like "does some given graph contain a triangle." "given a set of cities and travel times between them, is there a tour of length no more than some constant C," or "given a list of numbers, are any of them greater than 12."

An **instance** of a problem is a specific case of that problem: i.e. an instance of "does some given graph contain a triangle" is "does the complete graph K_{23} on 23 vertices contain a triangle."

Given a problem R, an **algorithm** is a step-by-step process that takes in an instance of a problem and outputs a solution. For example, an algorithm to solve "given a list of numbers, are any of them greater than 12" could be the process that takes a list of numbers, bubblesorts it, looks at the last element, and returns either true or false depending on whether that last element is greater than 12 or not.

Algorithms often create a **proof** when they are ran, that is used to output their true or false answer. For example, the algorithm we described for answering "given a list of numbers, are any of them greater than 12" creates a **proof** whenever it runs, by creating a sorted list, that demonstrates whether or not a number greater than 12 is in the list.

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 $O(n^k)$.

Problems in P, in a sense, are the ones we have reasonable hopes of computing for most relevant values of n. Granted, some polynomials (say $n^{1000000} + 2$) are going to be much worse than others to compute, but the argument we made above still holds — if we can run an algorithm in class P on some value of n, we can likely run it on $n + 1$ without much more effort.

NP is the following second class of problems:

Definition. A true-false 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 responds "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, NP is the set of problems that we can "quickly check solutions for." In a sense, pretty much any reasonable task that you'd ever ask anyone to solve falls in this category. This, heuristically speaking, is because of the following idea: if a task is remotely reasonable or useful, then solutions of it should be relatively easy to tell apart from nonsolutions (or you could just not bother finding a solution to your problem in the first place!)

5 P versus NP: an example

So, now you can understand the question at the heart of P versus NP:

Question 1. *Are the two classes P and NP different? In other words, does $P \neq NP$?*

This is a massive question, and we could easily devote, oh, say, the rest of our lives to studying it. But we only have twenty more minutes (probably) left in class; so instead, we're going to describe a famous question in NP!

Definition. A **latin square** of order n is a $n \times n$ array filled with n distinct symbols (by convention $\{1, \dots n\}$), such that no symbol is repeated twice in any row or column.

Example. Here are all of the latin squares of order 2:

$$\begin{bmatrix} 1 & 2 \\ 2 & 1 \end{bmatrix} \quad \begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix}.$$

Definition. A **partial latin square** of order n is a $n \times n$ array where each cell is filled with either blanks or symbols $\{1, \dots n\}$, such that no symbol is repeated twice in any row or column.

Example. Here are a pair of partial 4×4 latin squares:

$$\begin{bmatrix} & & & 4 \\ 2 & & & \\ 3 & 4 & & \\ 4 & 1 & 2 & \end{bmatrix} \qquad \begin{bmatrix} 1 & & & \\ & 1 & & \\ & & 1 & \\ & & & 2 \end{bmatrix}$$

The most obvious question we can ask about partial latin squares is the following: when can we complete them into filled-in latin squares? There are clearly cases where this is possible: the first array above, for example, can be completed as illustrated below.

$$\begin{bmatrix} & & & 4 \\ 2 & & & \\ 3 & 4 & & \\ 4 & 1 & 2 & \end{bmatrix} \mapsto \begin{bmatrix} 1 & 2 & 3 & 4 \\ 2 & 3 & 4 & 1 \\ 3 & 4 & 1 & 2 \\ 4 & 1 & 2 & 3 \end{bmatrix}$$

However, there are also clearly partial Latin squares that cannot be completed. For example, if we look at the second array

$$\begin{bmatrix} 1 & & & \\ & 1 & & \\ & & 1 & \\ & & & 2 \end{bmatrix},$$

we can pretty quickly see that there is no way to complete this array to a Latin square: any 4×4 Latin square will have to have a 1 in its last column somewhere, yet it cannot be in any of the three available slots in that last column, because there's already a 1 in those three rows.

Deciding whether a given partial Latin square is completeable to a Latin square is, practically speaking, a useful thing to be able to do. Consider the following simplistic model of a **router**:

- Setup: suppose you have a box with n fiber-optic cables entering it and n fiber-optic cables leaving it. On any of these cables, you have at most n distinct possible wavelengths of light that can be transmitted through that cable simultaneously. As well, you have some sort of magical/electronical device that is capable of "routing" signals from incoming cables to outgoing cables: i.e. it's a list of rules of the form (r, c, s), each of which send mean "send all signals of wavelength s from incoming cable r to outgoing cable s." These rules cannot conflict: i.e. if we're sending wavelength s from incoming cable r to outgoing cable s, we cannot also send s from r to t, for some other outgoing cable t. (Similarly, we cannot have two transmits of the form $\{(r, c, s), (r, t, s)\}$ or $\{(r, c, s), (t, c, s)\}$.)

- Now, suppose that your box currently has some predefined set of rules it would like to keep preserving: i.e. it already has some set of rules $\{(r_1, c_1, s_1), \ldots\}$. We can model this as a **partial Latin square**, by simply interpreting each rule (r, c, s) as "fill entry (r, c) of our partial Latin square with symbol s."

- With this analogy made, adding more symbols to our partial Latin square is equivalent to increasing the amount of traffic being handled by our router.

With this said, the NP question I'm interested in is the following:

Question 2. *Take a partial latin square P. Does it have a completion to a latin square L?*

This problem is in NP! To see this, simply note that "checking" any claimed method to complete a $n \times n$ partial Latin square P is quick: just take the claimed solution L, and

1. Go through each row of L and make sure that there are no repeated symbols. This takes n checks for each row and there are n rows in total, so this takes us n^2 steps in total.

2. Go through each column of L and make sure that there are no repeated symbols. This takes n checks for each column and there are n columns in total, so this takes us n^2 steps in total.

3. Go through each completed cell of P and check to make sure that P, L agree at that cell. Because P has at most $n \times n = n^2$ filled cells, this takes us at most n^2 many steps to check.

In total we took $O(n^2)$ many steps; therefore this process is a polynomial-time algorithm to check if P is completable!

There are many open research questions related to this problem. Namely, while completing a general Latin square is a problem in NP, there are many simpler problems, like

1. Completing a $n \times n$ partial Latin square that contains at most $n - 1$ filled cells, or
2. Completing a $n \times n$ partial Latin square that consists of 2 filled rows and columns, or
3. Completing a $n \times n$ partial Latin square where each row and column has at most $n/4$ nonblank cells, and no symbol is used more than $n/4$ many times in total

that we either know are in P (the first two) or don't know the answer to (the third one, which is a research problem I'm currently working on!)