Math 7h — Professor: Padraic Bartlett

Lecture 5: P versus NP

Week 5 — *UCSB 2013*

First, a joke:

Theorem. P = NP.

Proof. Simply set $N = 1$. □

Ok, bad joke. But a joke that raises a question: what **are** P and NP? When people talk about "P versus NP," what do they mean?

We answer this question in this talk.

1 Algorithms and Runtime

In week 2, we studied a series of **algorithms**: i.e. step-by-step procedures to solve various kinds of problems. For many of these algorithms, we looked at the **worst-case runtime** of these algorithms — i.e. given an input set of size n, we attempted to bound the number of steps or operations these algorithms would need to solve their problem.

In some cases, this was unbounded. Recall everyone's favorite sorting algorithm, **bogosort**:

Algorithm. Take as input some list $L = (l_1, \ldots l_n)$ of objects, along with some operation $\leq$ that can compare any two of these elements. Perform the following algorithm:

1. Check this list to see if it is already in order. (Formally: create some variable *loc*, initialize it at 1, and go through the list comparing elements l_{loc}, l_{loc+1}. If there's any pair that are out of order, stop searching and declare the list out of order; otherwise, continue until you've searched the whole list.)
2. If the list was in order, stop! You're done.
3. Otherwise, the list is out of order. Randomly select a permutation of our list, and reorder our list according to this permutation. Go to 1.

In the worst-case scenario, bogosort will never sort a list: it is possible, though vanishingly unlikely, that we never pick a permutation of our list that orders it.

We also described algorithms with demonstrably better worst-case runtimes than, um, forever. One such example was **bubblesort**:

Algorithm. Take as input some list $L = (l_1, \ldots l_n)$ of objects, along with some operation $\leq$ that can compare any two of these elements. Perform the following algorithm:

1. Create a integer variable *loc* and a boolean variable *didSwap*.
2. Set the variable *loc* to 1, and *didSwap* to $false$.
3. While the variable *loc* is $< n$, perform the following steps:
   (a) If $l_{loc} > l_{loc+1}$, swap the two elements l_{loc}, l_{loc+1} in our list and set *didSwap* to true.
   (b) Regardless of whether you swapped elements or not, add 1 to the variable *loc*, and go to 3.
4. If *didSwap* is $true$, then go to 2.
5. Otherwise, if *didSwap* is $false$, we went through our entire list and never found any pair of consecutive elements such that the second was larger than the first. Therefore, our list is sorted! Output our list.

We proved that bubblesort had a worst-case runtime of $O(n^2)$: i.e. there is some constant C such that if you feed bubblesort a list containing n elements, it will sort that list after at most Cn^2-many steps. (The precise value of C depends on your hardware and how you specifically implement bubblesort, which is why we usually omit this C and write $O(n^2)$. In practice for bubblesort, you can get C to be not much more than a half, with some clever optimization: try to do this!)

So: these are two algorithms to solve the same problem, one of which has a demonstrably shorter runtime than the other.

This raises the question in general: 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?

2 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.

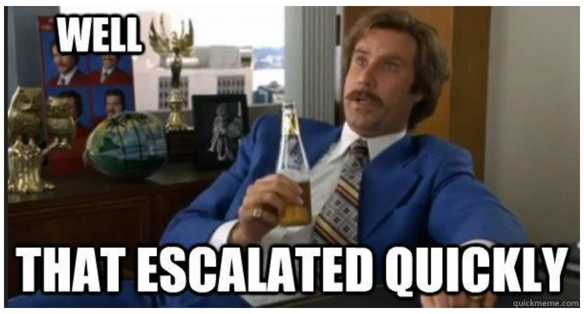
Runtime v. Input	10	20	30	40	50
n	$1 \cdot 10^{-5}$ sec.	$2 \cdot 10^{-5}$ sec.	$3 \cdot 10^{-5}$ sec.	$4 \cdot 10^{-5}$ sec.	$5 \cdot 10^{-5}$ sec.
n^2	$1 \cdot 10^{-4}$ sec.	$4 \cdot 10^{-4}$ sec.	$9 \cdot 10^{-4}$ sec.	$1.6 \cdot 10^{-3}$ sec.	$2.5 \cdot 10^{-3}$ sec.
n^3	$1 \cdot 10^{-3}$ sec.	$8 \cdot 10^{-3}$ 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.

3 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!)

The following theorem relates P to NP:

Theorem. Any problem in P is in NP.

Proof. Take any problem R in P. By definition, it has an algorithm A that given any instance of P, can create a solution in polynomial time.

We now claim that R is in NP. In other words, we claim that given any instance I and any proof S that claims I is true, we can quickly verify whether this proof actually corresponds to our instance being true.

We do this via the following algorithm:

1. Take our claimed proof S.
2. Set it on fire.
3. Now, take our polynomial-time algorithm A, that we have because our problem is in P.
4. Run this algorithm on our instance R.
5. In polynomial time, our algorithm will return either true or false. If it's true, then we have determined in polynomial time that our instance is true!
6. If it's false, then we have determined in polynomial time that our instance is false. Also, if it's false, then we don't feel guilty about setting our proof on fire.

Technically, the only steps we needed were 3-5. But 1,2 and 6 were satisfying, weren't they? □

4 P versus NP: an example

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

Conjecture. The two classes P and NP are different. In other words, 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!

Problem. Traveling Salesman. An **instance** of the Traveling Salesman problem consists of the following:

- A list $\{C_1, \ldots C_N\}$ of cities.
- A distance function $d(C_i, C_j)$ that outputs distances between these cities.
- A maximum distance D.

The **problem** itself is the following: given an instance of the Traveling Salesman problem, can you create a "tour" of these cities with distance no greater than D? In other words, can you create a path that visits each city once, starting and ending at the same city, that has total length no greater than D?

There is a relatively-simple algorithm to solve instance of the traveling salesman problem:

Algorithm.

List out all possible permutations of the list $\{C_1, \ldots C_N\}$ of cities.

For each permutation $(C'_1, \ldots C'_N)$ of this list, calculate the total tour distance

$$d(C'_n, C'_1) + \sum_{i=1}^{n} d(C'_i, C'_{i+1}).$$

If it is less than D, stop and output true, along with this permutation as a "proof."

Otherwise, continue through all of the permutations. If we get through all of the $N!$ permutations of the cities without finding a satisfactory tour, output false.

This is an algorithm, with runtime $O(n!)$. ($n!$, by the way, is exponential growth even though it may not look like it: it's actually asymptotically about $\sqrt{2\pi n} \cdot \left(\frac{n}{e}\right)^n$.) Which is kinda incredibly huge: i.e. 25! is greater than $1.5 \cdot 10^{25}$. If you could perform one step every microsecond, like before, a task that needed 25! steps would take about 4.75 years to complete. Factorial **bad**.

However, this problem **is** in NP! We prove this here:

Theorem. The traveling salesman problem is in NP.

Proof. A **proof** for the traveling salesman problem is an ordering $(C'_1, \ldots C'_N)$ of its cities. To check any proof, all we have to do is calculate

$$d(C'_n, C'_1) + \sum_{i=1}^{n} d(C'_i, C'_{i+1}),$$

which we can do with n pairwise addition steps, and then compare this to D with one more step. $n + 1$ is clearly a polynomial; so we've checked our proof in polynomial time! $\square$

There are many problems in NP. I close by mentioning one that I am interested in because my own research is connected to it:

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

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

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

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

We will talk a **lot** more about latin squares later.