NP and Latin Squares | Instructor: Padraic Bartlett

Lecture 1: P and NP

Week 4 | *Mathcamp 2014*

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 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:

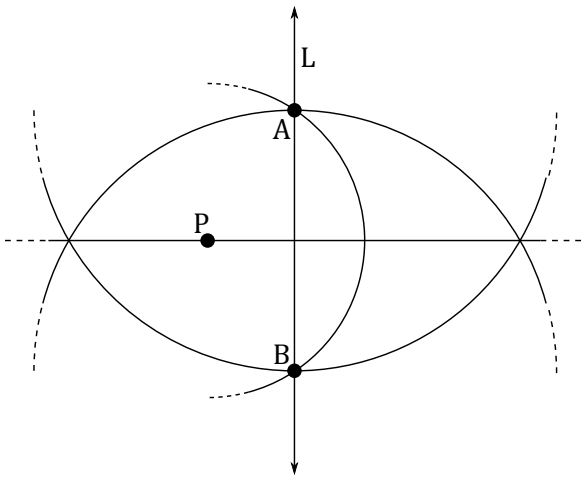
1. Pick any point on L. Call it A.
2. Draw a circle centered at P through A.
3. If this circle only intersects L at A and no other point, draw a line through A and P. This line is perpendicular to L.

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!

Another example of an algorithm 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 a number to itself, 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	–	–

An important class of algorithms (that we will not focus on in this class) are the set of **sorting algorithms**: algorithms that take in a list $L = (l_1, \ldots l_n)$ of objects, along with some method $\leqslant$ of comparing any two elements in our list. Given this initialization, a sorting algorithm will want to output an **ordering** of L: i.e. some permutation $(l'_1, \ldots l'_n)$ of the elements of L, so that $l'_1 \leqslant l'_2 \leqslant \ldots l'_n$.

One example, **bubblesort**, is a relatively classical and beautiful sorting algorithm:

Algorithm. Take as input some list $L = (l_1, \ldots l_n)$ of objects, along with some operation $\leqslant$ 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.

A sample run of this algorithm on the list $(1, 4, 3, 2, 5)$ is presented below:

run	step	*didSwap*	*loc*	L
1	2	$false$	1	$(1, 4, 3, 2, 5)$
1	3(a)	$false$	1	$(\mathbf{1}, \mathbf{4}, 3, 2, 5)$
1	3(b)	$false$	2	$(\mathbf{1}, \mathbf{4}, 3, 2, 5)$
1	3(a)	$true$	2	$(1, \mathbf{3}, \mathbf{4}, 2, 5)$
1	3(b)	$true$	3	$(1, \mathbf{3}, \mathbf{4}, 2, 5)$
1	3(a)	$true$	3	$(1, 3, \mathbf{2}, \mathbf{4}, 5)$
1	3(b)	$true$	4	$(1, 3, \mathbf{2}, \mathbf{4}, 5)$
1	3(a)	$true$	4	$(1, 3, 2, \mathbf{4}, \mathbf{5})$
1	3(b)	$true$	5	$(1, 3, 2, \mathbf{4}, \mathbf{5})$
1	4	$true$	5	$(1, 3, 2, 4, 5)$
2	2	$false$	1	$(1, 3, 2, 4, 5)$
2	3(a)	$false$	1	$(\mathbf{1}, \mathbf{3}, 2, 4, 5)$
2	3(b)	$false$	2	$(\mathbf{1}, \mathbf{3}, 2, 4, 5)$
2	3(a)	$true$	2	$(1, \mathbf{2}, \mathbf{3}, 4, 5)$
2	3(b)	$true$	3	$(1, \mathbf{2}, \mathbf{3}, 4, 5)$
2	3(a)	$true$	3	$(1, 2, \mathbf{3}, \mathbf{4}, 5)$
2	3(b)	$true$	4	$(1, 2, \mathbf{3}, \mathbf{4}, 5)$
2	3(a)	$true$	4	$(1, 2, 3, \mathbf{4}, \mathbf{5})$
2	3(b)	$true$	5	$(1, 2, 3, \mathbf{4}, \mathbf{5})$
2	4	$true$	5	$(1, 2, 3, 4, 5)$
3	2	$false$	1	$(1, 2, 3, 4, 5)$
3	3(a)	$false$	1	$(\mathbf{1}, \mathbf{2}, 3, 4, 5)$
3	3(b)	$false$	2	$(\mathbf{1}, \mathbf{2}, 3, 4, 5)$
3	3(a)	$false$	2	$(1, \mathbf{2}, \mathbf{3}, 4, 5)$
3	3(b)	$false$	3	$(1, \mathbf{2}, \mathbf{3}, 4, 5)$
3	3(a)	$false$	3	$(1, 2, \mathbf{3}, \mathbf{4}, 5)$
3	3(b)	$false$	4	$(1, 2, \mathbf{3}, \mathbf{4}, 5)$
3	3(a)	$false$	4	$(1, 2, 3, \mathbf{4}, \mathbf{5})$
3	3(b)	$false$	5	$(1, 2, 3, \mathbf{4}, \mathbf{5})$
3	4	$false$	5	$(1, 2, 3, 4, 5)$
3	5	–	–	$(1, 2, 3, 4, 5)$

One property we like to study in algorithms is the idea of **runtime**. For example, if we put the list $(2, 3, 4, 5, 1)$ into the bubblesort algorithm above, it will take five different passes through our list to sort this into the order $(1, 2, 3, 4, 5)$: specifically, each pass will move the symbol 1 one step over to the left and otherwise not change anything. In general, a list of the form $(2, 3, \ldots n, 1)$ will take n passes, and require us to go through roughly n^2 many steps (up to some smaller-order linear and constant bits of extra work we'll have to do.)

This, roughly speaking, is the worst-case scenario for bubblesort: after one run of bubblesort, we can see from examining the algorithm that the largest element will be in the last place in our list (because as soon as we get to it, we'll always swap it with successive elements until it gets to the last spot.) A second run insures that the second-largest ele-

ment gets to the second-largest spot, and so on/so forth: therefore, we know that n runs will always suffice.

We say that bubblesort has **worst-case** runtimes of $O(n^2)$, based on this analysis: roughly speaking, the worst bubblesort can do is take n passes, and therefore needs at most some constant $\cdot n^2$ many steps to sort a list. However, there are many lists on which bubblesort will perform markedly better than this: for example, if we have a list of the form $(n, 1, 2 \dots n-1)$, bubblesort will sort it in exactly one pass, i.e. in $O(n)$ many steps. So, while bubblesort may seem slow in general, it may actually be rather fast on certain kinds of lists! (Formally: we say that some quantity $f(n)$ is $O(n^2)$ if there is some constant M such that $\frac{f(n)}{n^2} \leqslant M$. Roughly speaking, this says that $f(n)$ grows "like" n^2, up to some fixed constant multiplier. This idea extends to $O(n), O(\log(n)$, and really $O(anything)$.)

Another sort, which is undoubtedly much more awful than bubblesort, is **bogosort**, which we define here:

Algorithm. Take as input some list $L = (l_1, \dots l_n)$ of objects, along with some operation $\leqslant$ 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.

This algorithm is akin to sorting a deck of cards by repeatedly (a) checking if it's in order, (b) throwing it up in the air if it's not, and returning to (a).

If L's elements are all distinct, bogosort has an expected runtime of $n!$; there are $n!$ many ways to permute the elements of our list, while exactly one of them is the correct sorting of L. Similarly, bogosort has a worst-case runtime of ∞; given any finite number of steps t, it's easily possible that our first t shuffles don't accidentally sort our list.

2 Algorithms and Runtime

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?

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 ... things grow a bit faster.

Roughly speaking: 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 — arguably one of the most important problems in modern science. We're probably not going to solve it in this class.

But we will try to understand it! To do this, let's find some examples of problems 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! □

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:

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

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 this class is focused on is the following:

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

We list a few additional famous problems in NP here:

- **SAT**: A **boolean variable** is simply some variable that can be assigned to either true or false. Given two boolean variables x, y, we can create various boolean expressions by combining them in various ways:
  - $x \wedge y$, pronounced "x and y," is the boolean expression that evaluates to **true** whenever both x **and** y are true, and **false** otherwise.
  - $x \vee y$, pronounced "x or y," is the boolean expression that evaluates to **true** whenever one of either x **or** y are true, and **false** otherwise. It bears noting that this is an "inclusive" or: i.e. if both x and y are true, then certainly *one* of x and y are true, and thus we say that $x \vee y$ is true.
  - $\neg x$, pronounced "not x", is true if x is false, and false if x is true.

  A **boolean formula** is just some long chain of boolean variables linked together by and's, or's and not's. For example, the formula below is a Boolean formula:

$$(x \wedge y) \vee (\neg(z \wedge (x \vee y))) \vee (a \wedge a).$$

A formula is called **satisfiable** if there is some way of assigning true or false to each of its variables, in such a way that the formula evaluates to true. For example, the formula above is satisfiable: set $x, y, z, a =$ true, which yields

$$\begin{aligned}(T \wedge T) \vee (\neg(T \wedge (T \vee T))) \vee (T \wedge T) &= T \vee (\neg(T \wedge T)) \vee T \\ &= T \vee (\neg T) \vee T \\ &= T \vee F \vee T \\ &= T.\end{aligned}$$

SAT is the following problem: given some boolean formula, is it satisfiable?

- **3SAT**: A boolean formula is said to be in **conjunctive normal form** if we can write it as the conjunction (and) of several clauses, where a clause is some disjunction (or) of several literals, and a literal is either a variable or its negation. For example, the following formula is in conjunctive normal form:

  $$(x \vee y \vee z) \wedge (\neg x \vee x) \wedge (z) \wedge (y \vee y \vee x).$$

  A boolean formula is said to be in **3-conjunctive normal form** if we can write it in conjunctive normal form, where each disjunction (or) contains precisely three literals. For example, the following formula is in 3-conjunctive normal form:

  $$(x \vee y \vee z) \wedge (\neg x \vee x \vee x) \wedge (a \vee a \vee a).$$

  3SAT is the following problem: given any boolean formula written in 3-conjunctive normal form, is it satisfiable?

- k-**Independent Set**: A graph $G = (V, E)$ has an **independent set of size** k if there is some subset of k vertices in G, no two of which are connected by an edge.

  Our problem, then, is the following question: given a graph $G = (V, E)$, does it have an independent set of size k?

- k-**Clique**: A graph $G = (V, E)$ has a **clique of size** k if there is some subset of k vertices in G, every two of which are connected by an edge.

  Our problem, then, is the following question: given a graph $G = (V, E)$, does it have a clique of size k?

This is a large collection of problems, many of which look quite different! In this sense, it may not seem obvious that these are all problems we should be grouping together — maybe some of them are easier to solve than others?

Hahahaha, ... no. Consider the following definition:

Definition. A problem P is called **NP-hard** if "solving it allows you to solve every other problem in NP," in the following sense:

- Take any problem R in NP.

- Any algorithm A_P that solves P can be turned into an algorithm that solves R, with an increase in runtime that is at most polynomial.

In other words, if you can solve an NP-hard problem in polynomial time, then you can solve **every NP problem** in polynomial time.

Theorem. (Cook-Levin) There is an NP-hard problem. In particular, there is an NP-hard problem contained within NP! Specifically, SAT and 3SAT are NP-hard.

(We call any problem that is both NP-hard and in NP a **NP-complete** problem.)

This might be, um, unexpected. After all, all of these problems look remarkably different! How can solving 3SAT allow you to solve **all** of them?

We're going to prove this tomorrow. Instead, what I want to spend the rest of class talking about is how to discover **more** things like this! Specifically, consider the following blueprint for showing that a problem is NP-hard:

- Take any problem P that we suspect is NP-hard.
- Take as well some other problem R that we know is NP-hard, like 3SAT.
- Suppose that we can "reduce" R to P, in the following sense: given any algorithm that solves P, suppose that we can "transform" this algorithm into a new algorithm that can solve R, with an increase in time that is at most polynomial. (The idea here is that if R can be reduced to P, then knowing how to solve P lets you solve R: i.e. the problem R is relatively easy if we understand P, and has therefore been "reduced" to essentially the same problem as P.)
- Then P is NP-hard! This is because given any algorithm that solves P, we can turn it into an algorithm that solves R, which (because R is NP-hard) we can turn into an algorithm that solves **any** NP problem, with at most polynomial increases in runtime.

5 NP-Hard: Results

We show a few problems are NP-hard here:

Theorem. Given a graph G, determining for which values of k this graph has an **independent set** of size k is NP-hard.

Proof. We reduce 3SAT to the independent set problem. Doing this will give us that the independent set problem is NP-hard, which is our claim.

To do this, we perform the following process:

- Take any formula F in 3SAT. We will describe a way to construct in polynomial time a graph G such that the following holds:

$$F \text{ is satisfiable} \Leftrightarrow G \text{ has an independent set of size } k.$$

- If we can do this, then given any algorithm that solves the k-independent set problem, we can solve any instance F of 3SAT by simply encoding F as a graph in polynomial time and then running our algorithm to decide if F is satisfiable.
- This is a reduction of 3SAT to the independent set problem, which shows that the independent set problem is NP-complete!

So: take any boolean formula F in 3-conjunctive normal form, with k clauses. Create a graph G as follows:

- Vertices: for each literal in a clause of F, create a vertex for that literal.
- Edges: Connect two vertices whenever one of the two following conditions hold:
  1. The two literals that these two vertices correspond to are from the same clause.
  2. The two literals that these two vertices correspond to are a variable and its negation (i.e. one is x and the other is $\neg x$, for some x).

We now observe that our formula with k clauses is satisfiable if and only if our graph has an independent set of size k. This is not too hard to see:

1. First, assume that our graph has an independent set of k variables. Then in our independent set, we must have one vertex from each clause, because all of the vertices corresponding to any given clause are connected with edges. Furthermore, we never have a vertex x and its negation $\neg x$ associated to variables in our independent set, because we connected all such pairs with edges.

   Look at the truth assignment that makes every literal equal to true in the independent set (i.e. picks out truth values for the variables so that each literal $x, \neg y, \neg z \ldots$ in our independent set is true.) This is consistent, in that because we have no edges between x's and $\neg x$'s, we don't try to assign both true or false to any given variable. This might leave some variables with unassigned truth values; set those to true as well.

   This truth assignment then satisfies F! To see why, simply notice that in every clause, we have insured that one literal at least is true. This means that each clause, as it is an "or" of three things, one of which is true, evaluates to true. Therefore, the entire formula is k true's "and-ed" together, which is clearly true.

2. Now, we go the other direction: we will show that if our formula is satisfiable, then our graph has an independent set of size k. This is easy: take any set of variables that satisfies our formula. Because our formula is satisfied, there is one literal from each formula that is true: take one such vertex for each clause. We then know that none of these vertices are linked by edges, as they come from different clauses and they all correspond to true (so we never picked connected pairs of the form $x, \neg x$.) Therefore this is an independent set with one element for each clause: i.e. an independent set of size k!

This completes our reduction proof. □

Theorem. Given a graph G, determining for which values of k this graph has a **clique** of size $k$is NP-hard.

Proof. We reduce the independent set problem to the clique problem. Doing this gives us that the clique problem is NP-hard, as claimed.

This is easier than before! To reduce the independent set problem to the clique problem, simply make the following observation:

A graph has an independent set of size $k \Leftrightarrow$ its complement graph[1] has a clique of size k.

Therefore, if we can quickly decide if a graph has a clique of size k, we can quickly decide if it has an independent set of size k, by simply constructing the complement graph (which we can do in polynomial time.)

Therefore, this process reduces the independent set problem to the clique problem! $\square$

[1]The complement graph $\overline{G}$ to a graph G is the graph with the same set of vertices as G, where we connect two vertices in $\overline{G}$ precisely when those two vertices are not connected in G.