6 P, NP, and friends

We've seen that if we want to make progress in complexity, then we need to talk about asymptotics: not which problems can be solved in 10 000 steps, but for which problems can instances of size n be solved in cn^2 steps as n goes to infinity? We met TIME($f(n)$), the class of all problems solvable in O($f(n)$) steps, and SPACE($f(n)$), the class of all problems solvable using O($f(n)$) bits of memory.

But if we *really* want to make progress, then it's useful to take an even coarser-grained view: one where we distinguish between polynomial and exponential time, but *not* between O(n^2) and O(n^3) time. From this remove, we think of any polynomial bound as "fast," and any exponential bound as "slow."

Now, I realize people will immediately object: what if a problem is solvable in polynomial time, but the polynomial is $n^{50\,000}$? Or what if a problem takes exponential time, but the exponential is $1.000\,000\,01^n$? My answer is pragmatic: if cases like that regularly arose in practice, then it would've turned out that we were using the wrong abstraction. But so far, it seems like we're using the right abstraction. Of the big problems solvable in polynomial time – matching, linear programming, primality testing, etc. – most of them really *do* have practical algorithms. And of the big problems that we think take exponential time – theorem-proving, circuit minimization, etc. – most of them really *don't* have practical algorithms. So, that's the empirical skeleton holding up our fat and muscle.

PETTING ZOO

It's now time to meet the most basic complexity classes – the sheep and goats of the Complexity Zoo.

http://youtu.be/f477FnTe1M0
http://youtu.be/4qAIPC7vG3Y

- **P** is the class of problems solvable by a Turing machine in polynomial time. In other words, **P** is the union, over all positive integers k, of TIME(n^k). (Note that, by "problem," we'll always mean *decision problem*: a problem where the inputs are n-bit strings and the outputs are either yes or no.)
- **PSPACE** is the class of problems solvable in polynomial space (but unlimited time). In other words, it's the union over all integers k of SPACE(n^k).
- **EXP** is the class of problems solvable in exponential time. In other words, it's the union over all integers k of TIME(2^{n^k}).

Certainly **P** is contained in **PSPACE**. I claim that **PSPACE** is contained in **EXP**. Why?

Right: a machine with n^k bits of memory can only go through 2^{n^k} different configurations before it either halts or else gets stuck in an infinite loop.

Now, **NP** is the class of problems for which, if the answer is yes, then there's a polynomial-size *proof* of that fact that you can *check* in polynomial time. (The NP stands for "Nondeterministic Polynomial," in case you were wondering.) I could get more technical, but it's easiest to give an example: say, I give you a 10 000-digit number, and I ask whether it has a divisor ending in 3. Well, answering that question might take a Long, Long Time™. But if your grad student finds such a divisor *for* you, then you can easily check that it works: you don't need to trust your student (always a plus).

I claim that **NP** is contained in **PSPACE**. Why?

Right: in polynomial space, you can loop over all possible n^k-bit proofs and check them one by one. If the answer is "yes," then one of the proofs will work, while if the answer is "no," then none of them will work.

Certainly **P** is contained in **NP**: if you can answer a question yourself, then someone else can convince you that the answer is yes (if it *is* yes) without even telling you anything.

Of course, a question arises of whether **P** *equals* **NP**. In other words, if you can *recognize* an answer efficiently, can you also *find* one efficiently? Maybe you've heard of this question before.

Look, this **P** versus **NP** question, what can I say? People like to describe it as "probably the central unsolved problem of theoretical computer science." That's a comical understatement. **P** vs. **NP** is one of the deepest questions that human beings have ever asked.

And not only that: it's one of the seven million-dollar prize problems of the Clay Math Institute![1] What an honor! Imagine: our mathematician friends have decided that **P** vs. **NP** is *as important* as the Hodge Conjecture, or even Navier–Stokes existence and smoothness! (Apparently, they weren't going to include it, until they asked around to make sure it was important enough.)

Dude. One way to measure **P** vs. **NP**'s importance is this. If **NP** problems were feasible, then mathematical creativity could be automated. The ability to check a proof would entail the ability to find one. Every Apple II, every Commodore, would have the reasoning power of Archimedes or Gauss. So by just programming your computer and letting it run, presumably you could immediately solve not only **P** vs. **NP**, but *also the other six Clay problems*. (Or five, now that Poincaré is down.)

But if that's the case, then why isn't it *obvious* that **P** doesn't equal **NP**? Surely, God wouldn't be so benign as to grant us these extravagant powers! Surely, our physicist-intuition tells us that brute-force search is unavoidable! (Leonid Levin told me that Feynman – the king, or possibly court jester, of physicist-intuition – had trouble even being convinced that **P** vs. **NP** was an open problem!)

Well, we certainly *believe* $\mathbf{P} \neq \mathbf{NP}$. Indeed, we don't even believe there's a general way to solve **NP** problems that's dramatically better than brute-force search through every possibility. But if you want to understand why it's so hard to *prove* these things, let me tell you something.

[1] See http://www.claymath.org/millennium/

http://youtu.be/f477FnTe1M0
http://youtu.be/4qAIPC7vG3Y

Let's say you're given an N-digit number, but instead of factoring it, you just want to know if it's prime or composite.

Or let's say you're given a list of freshmen, together with which ones are willing to room with which other ones, and you want to pair off as many willing roommates as you can.

Or let's say you're given two DNA sequences, and you want to know how many insertions and deletions are needed to convert the one sequence to the other.

Surely, these are fine examples of the sort of exponentially hard **NP** problem we were talking about! Surely, they, too, require brute-force search!

Except they don't. As it turns out, all of these problems have clever polynomial-time algorithms. *The central challenge any* ***P*** $\neq$ ***NP*** *proof will have to overcome is to separate the* ***NP*** *problems that really* ***are*** *hard from the ones that merely* ***look*** *hard*. I'm not just making a philosophical point. While there have been dozens of purported **P** $\neq$ **NP** proofs over the years, almost all of them could be rejected immediately for the simple reason that, if they worked, then they would rule out polynomial-time algorithms that we already know to exist.

So to summarize, there are problems like primality testing and pairing off roommates, for which computer scientists (often after decades of work) have been able to devise polynomial-time algorithms. But then there are other problems, like proving theorems, for which we don't know of any algorithm fundamentally better than brute-force search. But is that all we can say – that we have a bunch of these **NP** problems, and for some of them, we've found a fast algorithm and for others, we haven't?

As it turns out, we can say something much more interesting than that. We can say that *almost all of the "hard" problems are the* ***same*** *"hard" problem in different guises* – in the sense that, if we had a polynomial-time algorithm for any one of them, then we'd also have polynomial-time algorithms for all the rest. This is the upshot of the

theory of **NP**-completeness, which was created in the early 1970s by Cook, Karp, and Levin.

The way it goes is, we define a problem B to be "**NP**-hard" if any **NP** problem can be efficiently reduced to B. What the hell does that mean? It means that, *if* we had an oracle to immediately solve problem B, *then* we could solve any **NP** problem in polynomial time.

That gives one notion of reduction, which is called Cook reduction. There's also a weaker notion of reduction, which is called Karp reduction. In a Karp reduction from problem A to problem B, we insist that there should be a polynomial-time algorithm that transforms any instance of A to an instance of B having the same answer.

What's the difference between Cook and Karp?

Right: with a Cook reduction, in solving problem A we get to call the oracle for problem B more than once. We can even call the oracle *adaptively* – that is, in ways that depend on the outcomes of the previous calls. A Karp reduction is weaker in that we don't allow ourselves these liberties. Perhaps surprisingly, almost every reduction we know of is a Karp reduction – the full power of Cook reductions is rarely needed in practice.

Now, we say a problem is **NP**-complete if it's both **NP**-hard and in **NP**. In other words, **NP**-complete problems are the "hardest" problems in **NP**: the problems that single-handedly capture the difficulty of every other **NP** problem. As a first question, is it obvious that **NP**-complete problems even *exist*?

I claim that it *is* obvious. Why?

Well, consider the following problem, called DUH: we're given a polynomial-time Turing machine M, and we want to know if there exists an n^k-bit input string that causes M to accept. I claim that any instance of any **NP** problem can be converted, in polynomial time, into a DUH instance having the same answer. Why? Well, DUH! Because that's what it *means* for a problem to be in **NP**!

The discovery of Cook, Karp, and Levin was not that there *exist* **NP**-complete problems – that's obvious – but rather that many *natural* problems are **NP**-complete.

The king of these natural **NP**-complete problems is called 3-Satisfiability, or 3SAT. (How do I know it's the king? Because it appeared on the TV show *NUMB3RS*.) Here we're given n Boolean variables, $x_1, \ldots, x_n$, as well as a set of logical constraints called *clauses* that relate at most three variables each:

x_2 or x_5 or not(x_6)

not(x_2) or x_4

not(x_4) or not(x_5) or x_6

....

Then the question is whether there's some way to set the variables $x_1, \ldots, x_n$ to TRUE or FALSE, in such a way that every clause is "satisfied" (that is, every clause evaluates to TRUE).

It's obvious that 3SAT is *in* **NP**. Why? Right: Because if someone gives you a setting of $x_1, \ldots, x_n$ that works, it's easy to check that it works!

Our goal is to prove that 3SAT is **NP**-complete. What will that take? Well, we need to show that, if we had an oracle for 3SAT, then we could use it to solve not only 3SAT in polynomial time but also *any* **NP** problem whatsoever. That seems like a tall order! Yet in retrospect, you'll see that it's almost a triviality.

The proof has two steps. Step 1 is to show that, if we could solve 3SAT, then we could solve a more "general" problem called CircuitSAT. Step 2 is to show that, if we could solve CircuitSAT, then we could solve any **NP** problem.

In CircuitSAT, we're given a Boolean circuit and . . . wait, listen up, engineers: in computer science, a "circuit" *never* has loops! Nor does it have resistors or diodes or anything weird like that. For us,

a circuit is just an object where you start with n Boolean variables $x_1, \ldots, x_n$, and then you can repeatedly define a new variable that's equal to the AND, OR, or NOT of variables that you've previously defined. Like so:

$$x_{n+1} := x_3 \text{ or } x_n$$

$$x_{n+2} := \text{not}(x_{n+1})$$

$$x_{n+3} := x_1 \text{ and } x_{n+2}$$

$$\ldots.$$

We designate the last variable in the list as the circuit's "output." Then the goal, in CircuitSAT, is to decide whether there's a setting of $x_1, \ldots, x_n$ such that the output is TRUE.

I claim that, if we could solve 3SAT, then we could also solve CircuitSAT. Why?

Well, all we need to do is notice that every CircuitSAT instance is really a 3SAT instance in disguise! Every time we compute an AND, OR, or NOT, we're relating one new variable to one or two old variables. And any such relationship can be expressed by a set of clauses involving at most three variables each. So, for example,

$$x_{n+1} := x_3 \text{ or } x_n$$

becomes

$$x_{n+1} \text{ or not}(x_3)$$

$$x_{n+1} \text{ or not}(x_n)$$

$$\text{not}(x_{n+1}) \text{ or } x_3 \text{ or } x_n$$

So, that was Step 1. Step 2 is to show that, if we can solve CircuitSAT, then we can solve *any* **NP** problem.

Alright, so consider some instance of an **NP** problem. Then by the definition of **NP**, there's a polynomial-time Turing machine M such that the answer is "yes" if and only if there's a polynomial-size witness string w that causes M to accept.

Now, given this Turing machine M, our goal is to create a circuit that "mimics" M. In other words, we want there to exist a setting of the circuit's input variables that makes it evaluate to TRUE, *if and only if* there exists a string w that causes M to accept.

How do we achieve that? Simple: by *defining a whole buttload of variables!* We'll have a variable that equals TRUE if and only if the 37th bit of M's tape is set to '1' at the 42nd time step. We'll have another variable that equals TRUE if and only if the 14th bit is set to '1' at the 52nd time step. We'll have another variable that equals TRUE if and only if M's tape head is in the 15th internal state and the 74th tape position at the 33rd time step. Well, you get the idea.

Then, having written down this buttload of variables, we write down a shitload of logical relations between them. If the 17th bit of the tape is '0' at the 22nd time step, and the tape head is nowhere near the 17th bit at that time, then the 17th bit will still be '0' at the 23rd time step. If the tape head is in internal state 5 at the 44th time step, and it's reading a '1' at that time step, and internal state 5 transitions to internal state 7 on reading a '1', then the tape head will be in internal state 7 at the 45th time step. And so on, and so on. The only variables that are left unrestricted are the ones that constitute the string *w* at the first time step.

The key point is that, while this *is* a very large buttload of variables and relations, it's still only a *polynomial* buttload. We therefore get a polynomially large CircuitSAT instance, which is satisfiable if and only if there exists a *w* that causes M to accept.

We've just proved the celebrated Cook–Levin Theorem: that 3SAT is **NP**-complete. This theorem can be thought of as the "initial infection" of the **NP**-completeness virus. Since then, the virus has spread to *thousands* of other problems. What I mean is this: if you want to prove that your favorite problem is **NP**-complete, all you have to do is prove it's as hard as some other problem that's *already* been proved **NP**-complete. (Well, you also have to prove that it's *in* **NP**, but that's usually trivial.) So there's a rich-get-richer effect: the more problems

that have already been proved **NP**-complete, the easier it is to induct a new problem into the club. Indeed, proving problems **NP**-complete had become so routine by the 1980s or 1990s, and people had gotten so good at it, that (with rare exceptions) the two main complexity conferences STOC and FOCS stopped publishing yet more **NP**-completeness proofs.

I'll just give you a tiny sampling of some of the earliest problems that were proved **NP**-complete:

- **Map Colorability:** Given a map, can you color every country red, green, or blue, in such a way that no two neighboring countries are colored the same? (Interestingly, if you're only allowed to use *two* colors, then it's easy to decide whether or not such a coloring is possible – why? On the other hand, if you're allowed *four* colors, then it always is possible, at least for maps drawn in the plane – that's the famous Four-Color Theorem. So then the problem is again easy. Only with three colors is the problem **NP**-complete.)
- **Clique:** Given a set of N high-school students, together with which ones will sit at a cafeteria table with which other ones, is there a "clique" of $N/3$ students who will all sit at a table with each other?
- **Packing:** Given a set of boxes of specified dimensions, can you fit them into the trunk of your car?

Etc., etc., etc.

To reiterate: although these problems might look unrelated, they're actually the same problem in different costumes. If any *one* of them has an efficient solution, then *all* of them do, and $\mathbf{P} = \mathbf{NP}$. If any one of them *doesn't* have an efficient solution, then *none* of them do, and $\mathbf{P} \neq \mathbf{NP}$. To prove $\mathbf{P} = \mathbf{NP}$, it's enough to show that *some* **NP**-complete problem (no matter which one) has an efficient solution. To prove $\mathbf{P} \neq \mathbf{NP}$, it's enough to show that some **NP**-complete problem has *no* efficient solution. One for all and all for one.

So, there are the **P** problems, and then there are the **NP**-complete problems. Is there anything in between? (You should be used to this

sort of "intermediate" question by now – we saw it both in set theory and in computability theory!)

If **P** = **NP**, then **NP**-complete problems are **P** problems, so obviously the answer is no.

But what if **P** ≠ **NP**? In that case, a beautiful result called Ladner's Theorem says that there must be "intermediate" problems between **P** and **NP**-complete: in other words, problems that are in **NP**, but neither **NP**-complete nor solvable in polynomial time.

How would we create such an intermediate problem? Well, I'll give you the idea. The first step is to define an extremely slow-growing function t. Then, given a 3SAT instance F of size n, the problem will be to decide whether F is satisfiable *and* $t(n)$ is odd. In other words: if $t(n)$ is odd, then solve the 3SAT problem, while if $t(n)$ is even, then always output "no."

If you think about what we're doing, we're alternating long stretches of an **NP**-complete problem with long stretches of nothing! Intuitively, each stretch of 3SAT should kill off another polynomial-time algorithm for our problem, where we use the assumption that **P** ≠ **NP**. Likewise, each stretch of nothing should kill off another **NP**-completeness reduction, where we again use the assumption that **P** ≠ **NP**. This ensures that the problem is neither in **P** nor **NP**-complete. The main technical trick is to make the stretches get longer at an exponential rate. That way, given an input of size n, we can simulate the whole iterative process up to n in time polynomial in n. That ensures that the problem is still in **NP**.

Besides **P** and **NP**, another major complexity class is **coNP**: the "complement" of **NP**. A problem is in **coNP** if a "no" answer can be checked in polynomial time. To every **NP**-complete problem, there's a corresponding **coNP**-complete problem. We've got *un*satisfiability, map *non*colorability, etc.

Now, why would anyone bother to define such a stupid thing? Because then we can ask a new question: *does* ***NP*** *equal* ***coNP***? In other words: if a Boolean formula is unsatisfiable, is there at least a

http://youtu.be/f477FnTe1M0
http://youtu.be/4qAIPC7vG3Y

short *proof* that it's unsatisfiable, even if finding the proof would take exponential time? Once again, the answer is that we don't know.

Certainly, if **P** = **NP**, then **NP** = **coNP**. (Why?) On the other hand, the other direction isn't known: it could be that **P** ≠ **NP** but still **NP** = **coNP**. So if proving **P** ≠ **NP** is too easy, you can instead try to prove **NP** ≠ **coNP**!

This seems like a good time to mention a special complexity class, a class we quantum computing people know and love: **NP** ∩ **coNP**.

This is the class for which either a yes answer *or* a no answer has an efficiently checkable proof. As an example, consider the problem of factoring an integer into primes. Over the course of my life, I must've met at least two dozen people who "knew" that factoring is **NP**-complete, and therefore that Shor's algorithm – since it lets us factor on a quantum computer – also lets us solve **NP**-complete problems on a quantum computer. Often these people were supremely confident of their "knowledge."

Before we look into the possible **NP**-completeness of factoring, let me at least explain why I feel that factoring is not in **P**. Dare I say that the reason is that no one can solve it efficiently in practice? Though it's not a good argument, people are certainly counting on it not being in **P**. Admittedly, we don't have as strong a reason to believe that factoring is not in **P** as we do to believe that **P** ≠ **NP**. It's even a semirespectable opinion to say that maybe factoring is in **P**, and that we just don't know enough about number theory to prove it. If you think about it for two seconds, you'll realize that factoring has profound differences from the known **NP**-complete problems. If I give you a Boolean formula, it might have zero satisfying truth assignments, it might have one, or it might have 10 trillion. You simply don't know, a priori. But if I give you a 5000-digit integer, you probably won't know its factorization into primes, but you'll know that has *one and only one* factorization. (I think some guy named Euclid proved that a while ago.) This already tells us that factoring is somehow "special": that,

unlike what we believe about the **NP**-complete problems, factoring has some *structure* that algorithms could try to exploit. And, indeed, algorithms *do* exploit it: we know of a classical algorithm called the Number Field Sieve, which factors an n-bit integer in roughly $2^{n^{1/3}}$ steps, compared to the $\sim 2^{n/2}$ steps that would be needed for trying all possible divisors. (Why only $\sim 2^{n/2}$ steps, instead of $\sim 2^n$?) And, of course, we know of Shor's algorithm, which factors an n-bit integer in $\sim n^2$ steps on a quantum computer: that is, in quantum polynomial time. Contrary to popular belief, we *don't* know of a quantum algorithm to solve **NP**-complete problems in polynomial time. If such an algorithm existed, it would have to be dramatically different from Shor's algorithm.

But can we pinpoint just *how* factoring differs from the known **NP**-complete problems, in terms of complexity theory? Yes, we can. First of all, in order to make factoring a decision (yes-or-no) problem, we need to ask something like this: given a positive integer N, does N have a prime factor whose last digit is 7? I claim that this problem is not merely in **NP**, but in **NP** $\cap$ **coNP**. Why? Well, suppose someone gives you the prime factorization of N. There's only one of them. So if there *is* a prime factor whose last digit is 7, then you can verify that, and if there's *no* prime factor whose last digit is 7, then you can also verify *that*.

You might say, "but how do I know that I really was given the prime factorization? Sure, if someone gives me a bunch of numbers, I can check that they multiply to N, but how do I know they're prime?" For this, you'll have to take on faith something that I told you earlier: that if you just want to know whether a number is prime or composite, and not what its factors are, then you can do that in polynomial time. OK, so if you accept that, then the factoring problem is in **NP** $\cap$ **coNP**.

From this, we can conclude that, *if* factoring were **NP**-complete, then **NP** would equal **coNP**. (Why?) Since we don't believe **NP** = **coNP**, this gives us a strong indication (though not a proof) that, all those people I told you about notwithstanding, factoring is *not*

NP-complete. If we accept that, then only two possibilities remain: either factoring is in **P**, or else factoring is one of those "intermediate" problems whose existence is guaranteed by Ladner's Theorem. Most of us incline toward the latter possibility – though not with as much conviction as we believe $\mathbf{P} \neq \mathbf{NP}$.

Indeed, for all we know, it could be the case that $\mathbf{P} = \mathbf{NP} \cap \mathbf{coNP}$ but still $\mathbf{P} \neq \mathbf{NP}$. (This possibility would imply that $\mathbf{NP} \neq \mathbf{coNP}$.) So, if proving $\mathbf{P} \neq \mathbf{NP}$ and $\mathbf{NP} \neq \mathbf{coNP}$ are *both* too easy for you, your next challenge can be to prove $\mathbf{P} \neq \mathbf{NP} \cap \mathbf{coNP}$!

If **P**, **NP**, and **coNP** aren't enough to rock your world, you can generalize these classes to a giant teetering mess that we computer scientists call the *polynomial hierarchy*.

Observe that you can put any **NP** problem instance into the form

> *Does there exist an n-bit string X such that A(X)=1?*

Here A is a function computable in polynomial time.

Likewise, you can put any **coNP** problem into the form

> *Does A(X)=1 for every X?*

But what happens if you throw in another quantifier, like so?

> *Does there exist an X such that for every Y, A(X,Y)=1?*
>
> *For every X, does there exist a Y such that A(X,Y)=1?*

Problems like these lead to two new complexity classes, which are called $\Sigma_2\mathbf{P}$ and $\Pi_2\mathbf{P}$, respectively. $\Pi_2\mathbf{P}$ is the "complement" of $\Sigma_2\mathbf{P}$, in the same sense that **coNP** is the complement of **NP**. We can also throw in a third quantifier:

> *Does there exist an X such that for every Y, there exists a Z such that A(X,Y,Z)=1?*
>
> *For every X, does there exist a Y such that for every Z, A(X,Y,Z)=1?*

This gives us $\Sigma_3\mathbf{P}$ and $\Pi_3\mathbf{P}$, respectively. It should be obvious how to generalize this to $\Sigma_k\mathbf{P}$ and $\Pi_k\mathbf{P}$ for any larger k. (As a side note, when $k = 1$, we get $\Sigma_1\mathbf{P} = \mathbf{NP}$ and $\Pi_1\mathbf{P} = \mathbf{coNP}$. Why?) Then taking the union of these classes over all positive integers k gives us the polynomial hierarchy **PH**.

The polynomial hierarchy really is a substantial generalization of **NP** and **coNP** – in the sense that, even if we had an oracle for **NP**-complete problems, it's not at all clear how we could use it to solve (say) $\Sigma_2\mathbf{P}$ problems. On the other hand, just to complicate matters further, I claim that if $\mathbf{P} = \mathbf{NP}$, then the whole polynomial hierarchy would collapse down to **P**! Why?

Right: if $\mathbf{P} = \mathbf{NP}$, then we could take our algorithm for solving **NP**-complete problems in polynomial time, and modify it to *call itself as a subroutine*. And that would let us "flatten **PH** like a steamroller": first simulating **NP** and **coNP**, then $\Sigma_2\mathbf{P}$ and $\Pi_2\mathbf{P}$, and so on through the entire hierarchy.

Likewise, it's not hard to prove that, if $\mathbf{NP} = \mathbf{coNP}$, then the entire polynomial hierarchy collapses down to **NP** (or in other words, to **coNP**). If $\Sigma_2\mathbf{P} = \Pi_2\mathbf{P}$, then the entire polynomial hierarchy collapses down to $\Sigma_2\mathbf{P}$, and so on. If you think about it, this gives us a whole infinite sequence of generalizations of the $\mathbf{P} \neq \mathbf{NP}$ conjecture, each one "harder" to prove than the last. Why do we care about these generalizations? Because often, we're trying to study conjecture BLAH, and we can't prove that BLAH is true, and we can't *even* prove that if BLAH were false then **P** would equal **NP**. But – and here's the punchline – we *can* prove that if BLAH were false, then the polynomial hierarchy would collapse to the second or the third level. And this gives us some sort of evidence that BLAH is true.

Welcome to complexity theory!

Since I talked about how lots of problems have nonobvious polynomial-time algorithms, I thought I should give you at least one example. So, let's do one of the simplest and most elegant in all of

computer science – the so-called Stable Marriage Problem. Have you seen this before? You haven't?

Alright, so we have N men and N women. Our goal is to marry them off. We assume for simplicity that they're all straight. (Marrying off gays and lesbians is technically harder, though also solvable in polynomial time!) We also assume, for simplicity and with much loss of generality, that everyone would rather be married than single.

So, each man ranks the women, in order from his first to last choice. Each woman likewise ranks the men. There are no ties.

Obviously, not every man can marry his first-choice woman, and not every woman can marry her first-choice man. Life sucks that way.

So, let's try for something weaker. Given a way of pairing off the men and women, say that it's *stable* if *no man and woman who aren't married to each other both prefer each other to their spouses*. In other words, you might despise your husband, but no man who you like better than him likes you better than his wife, so you have no incentive to leave. This is the, um, desirable property that we call "stability."

Now, given the men's and women's stated preferences, our goal as matchmakers is to find a stable way of pairing them off. Matchmaker, matchmaker, make me a match, find me a find, catch me a catch, etc.

First obvious question: does there always exist a stable pairing of men and women? What do you think? Yes? No? As it turns out, the answer is yes, but the easiest way to prove it is just to give an algorithm for *finding* the pairing!

So, let's concentrate on the question of how to find a pairing. In total, there are $N!$ ways of pairing off men with women. For the soon-to-be-newlyweds' sake, we hope we won't have to search through all of them.

Fortunately, we won't. In the early 1960s, Gale and Shapley invented a polynomial-time – in fact *linear*-time – algorithm to solve

http://youtu.be/f477FnTe1M0
http://youtu.be/4qAIPC7vG3Y

this problem. And the beautiful thing about this algorithm is, it's exactly what you'd come up with from reading a Victorian romance novel. Later they found out that the same algorithm had been *in use* since the 1950s – not to pair off men with women, but to pair off medical-school students with hospitals to do their residencies in. Indeed, hospitals and medical schools are still using a version of the algorithm today.

But back to the men and women. If we want to pair them off by the Gale–Shapley algorithm, then as a first step, we need to break the symmetry between the sexes: which sex "proposes" to the other? This being the early 1960s, you can guess how that question was answered. The men propose to the women.

So, we loop through all the men. The first man proposes to his first-choice woman. She provisionally accepts him. Then the next man proposes to his first-choice woman. She provisionally accepts *him*, and so on. But what happens when a man proposes to a woman who's already provisionally accepted another man? She chooses the one she prefers, and boots the other one out! Then, the next time we come around to that man in our loop over the men, he'll propose to his *second*-choice woman. And if she rejects him, then the next time we come around to him he'll propose to his third-choice woman. And so on, until everyone is married off. Pretty simple, huh?

First question: why does this algorithm terminate in linear time?

Right: because each man proposes to a given woman at most once. So the total number of proposals is at most N^2, which is just the amount of memory we need to write down the preference lists in the first place.

Second question: when the algorithm does terminate, why is everyone married off?

Right: because if they weren't, then there'd be some woman who'd never been proposed to, and some man who'd never proposed to her. But this is impossible. Eventually, the man no one else wants will cave in, and propose to the woman no one else wants.

http://youtu.be/f477FnTe1M0
http://youtu.be/4qAIPC7vG3Y