Lecture 24

NP intuitions and Ladner's Theorem

In this lecture, we will start by reviewing some intuitions for NP. We will also show that if P = NP, then the witnesses for NP problems can be found in polynomial time.

Finally, will prove Ladner's theorem. This theorem says that if P ≠ NP, then there are some problems in NP that are not in P but also not NP-complete. Such problems are called "NP intermediate".

24.1 From decision to search

Let A be a laungage in NP. We know there is a polynomial-time verifier V for A, such that a string x is in A if and only if there is a polynomially-sized witness w such that $V(\langle x, w\rangle)$ accepts.

Now suppose that P = NP. This implies that there is a polynomial-time algorithm for deciding if a string x is in A. But does it also mean there is a polynomial-time algorithm for *finding* the polynomially-sized witness w for which $V(\langle x, w\rangle)$ accepts? Such a witness must exist if x is in A, but can we find it?

It turns out that the answer is yes, Using the following trick. Let $k, c \in \mathbb{N}$ be such that each string $x \in A$ has a witness w of size at most $|x|^k + c$ which is accepted by V. Consider the language

$$B = \{\langle x, y\rangle : \exists z \in \Sigma^* \text{ such that } |yz| \leq |x|^k + c \text{ and } V(\langle x, yz\rangle) \text{ accepts}\}.$$

We claim this language B is in NP. To see this, we define a verifier V' for B. On input $\langle s, z\rangle$, the verifier V' will check if s has the format $s = \langle x, y\rangle$, rejecting otherwise. If s is correctly formatted, V' will then check if $|yz| \leq |x|^k + c$, and if so, it will run $V(\langle x, yz\rangle)$, and accept if this computation accepts. This takes only polynomial time in the input to V. It is easy to see that this is a valid polynomial-time verifier: if $\langle x, y\rangle \in B$, then a witness z which causes $V'((\langle\langle x, y\rangle, z\rangle)$ to accept must exist, and it must have size at most $|x|^k + c$, which is polynomial in the input size; conversely, if $\langle x, y\rangle \notin B$, then such a witness z cannot exist.

Since we are assuming P = NP and since $B \in$ NP, we conclude that $B \in$ P. This means there is a polynomial-time algorithm for deciding B, say M_B. We now describe a polynomial-time Turing machine M which takes as input $x \in A$ and outputs a witness w such that $V(\langle x, w\rangle)$ accepts.

Assume for simplicity that the alphabet is $\Sigma = \{0, 1\}$. On input x, the machine M will first run $M_B(\langle x, \epsilon\rangle)$, which is equivalent to checking if x is really in A; if this rejects, M will print out "x is not in A so no witness exists". If it accepts, M will run $V(\langle x, \epsilon\rangle)$ to check if ϵ is a witness for x; if so, M can output that as a witness.

Next, M will run $M_B(\langle x, 0\rangle)$ and $M_B(\langle x, 1\rangle)$. Since $x \in A$, x has some witness $w \neq \epsilon$, and this witness must start with either 0 or 1; hence at least one of $M_B(\langle x, 0\rangle)$ and $M_B(\langle x, 1\rangle)$ must accept.

If $M_B(\langle x, 0\rangle)$ accepts, for example, then we know a valid witness for x exists that starts with the bit 0. The next step of M will be to run $V(\langle x, 0\rangle)$ to check whether $w = 0$ is a witness for x; if so, it could output it.

After this, M will run $M_B(\langle x, 00\rangle)$ and $M_B(\langle x, 01\rangle)$ to check whether the second bit of w is 0 or 1. M will keep going in this way, increasing the size of the initial segment y of the true witness w it is looking for. Eventually, after at most $|x|^k + c$ iterations, M will find a witness w such that $V(\langle x, w\rangle)$ accepts, and M will then output w.

This machine M finds a valid witness w for x, assuming one exists. What is the running time of M? Well, M simply makes $O(|x|^k)$ calls to M_B and to V, and the input sizes to M_B and to V are at most $O(|x|^k)$ each time. Since M_B and V both run in polynomial time, the entire running time of M is still polynomial in the input size $|x|$.

We conclude that if $\mathsf{P} = \mathsf{NP}$, then not only could we *decide* languages in NP in polynomial time, we could even find specific witnesses that cause the verifier to accept a given string.

24.2 NP as a class of puzzles

We can think of P as the class of problems that are efficiently solvable. How should we think of NP?

One way to think of it is as a class of *puzzles*, to which a solution can be easily verified. For example, Let A be the language of all Sudoku puzzles on a generalized, $n \times n$ board. Each string $x \in A$ encodes a Sudoku puzzle, which means it is an $n \times n$ grid which is partially filled with numbers 1 through n such that there is a way to fill the rest of the cells of the grid to make a valid Sudoku solution.

Deciding if a language is in A corresponds to deciding if a given Sudoku puzzle has a valid solution (since A only contains the solvable Sudoku puzzles). We can easily construct a polynomial-time verifier for A: the verifier V will take in a puzzle x and a string w representing a solution to x, and it will simply check that w is a valid solution to the puzzle. This checking process is straightforward and can be done efficiently. Hence A is in NP.

If $\mathsf{P} = \mathsf{NP}$, it would mean we have an efficient algorithm for deciding if a given Sudoku puzzle is solvable. But from the previous section, we know it would also mean we have an efficient algorithm for *solving* Sudoku puzzles (since solving a given puzzle x is equivalent to finding a witness w accepted by V). So if $\mathsf{P} = \mathsf{NP}$, we could solve any Sudoku puzzle in polynomial time.

The neat thing is that this argument didn't use anything about how a Sudoku puzzle works at all. The only thing we needed is that a solution to the puzzle can be easily verified by a polynomial-time program V. In other words, if $\mathsf{P} = \mathsf{NP}$, then *all* possible puzzles can be efficiently solved in polynomial time (or at least, all puzzles for which it is possible to efficiently *verify* a solution; this is a requirement in order for the puzzle language to be in NP in the first place).

24.2.1 Automated theorem-proving

Here is another example of an interesting language in NP. Consider the language B of all mathematical statements s which have a proof of some polynomial size, say $|s|^{10} + 1000$. Determining whether a mathematical statement is in B would let us know whether the statement has a reasonably short proof. (Note that if a statement has no short proof, we would probably never find a proof of it anyway, since the shortest proof would be too long for any human to read.)

The language B is in NP, because for each string $s \in B$ there is a witness: the witness w would be the mathematical proof that s is a valid statement. Of couse, this requires us to have a verifier V which can check whether a given proof w really is a proof of a given statement s, but such verification algorithms exist (we can turn mathematics into computer-checkable format).

If $\mathsf{P} = \mathsf{NP}$, then we could determine whether a given mathematical statement has a not-too-long proof. Moreover, we could even find the witness w, and hence we could even find a proof for any given mathematical statement to which a reasonably-sized proof exists – all in polynomial time. This would let us automate mathematics.

24.2.2 More about P versus NP

Given that the statement $\mathsf{P} = \mathsf{NP}$ implies that all possible puzzles are efficiently solvable and that mathematics is efficiently automatable, isn't it obvious that $\mathsf{P} \neq \mathsf{NP}$? It may seem obvious, but we don't know how to prove it.

The issue with proving it is that we can't rule out the possibility that a really clever algorithm can look at the code of a verifier V, and immediately discern whether a given string x will have a witness w that V accepts. Now, the motto of this course would say that this is impossible; recall that our motto was

> You can never find general mechanical means
> for prediction the acts of computing machines
> (at least, not any better than just running the machine).

We could prove various formalizations of this motto, including the undecidability of the Halting problem and the Time Hierarchy theorem. The P versus NP question can also be viewed as a formalization of this theorem, because we are saying that there is no way to know whether $V(x, \cdot)$ accepts for some string w without actually running $V(x, w)$ for every candidate witness w (and there are exponentially many of those). However, this time, we cannot prove this particular formalization of the motto. Most people conjecture that the motto still holds, which would mean that $\mathsf{P} \neq \mathsf{NP}$.

In other words, you can think of the task of proving $\mathsf{P} \neq \mathsf{NP}$ as the task of showing that there are no shortcuts to deciding whether a program $V(x, \cdot)$ accepts for some string w; one must try to run $V(x, w)$ on all possible appropriately-sized strings w. Unfortunately, we cannot prove that there are no shortcuts: maybe a very clever algorithm can use the code of V to decide whether such a witness w exists without trying all of them.

24.3 Ladner's theorem

We now turn to Ladner's theorem. Ladner's theorem essentially says that there must be languages in NP that are not in P but also not NP-complete. Such languages are called NP-intermediate.

Actually, there is a problem with such a statement: we don't know that $\mathsf{P} \neq \mathsf{NP}$. If it turns out that $\mathsf{P} = \mathsf{NP}$, then every language in NP (except for 2 of them) will be NP-complete (since all but 2 languages are P-hard), and also every language in NP will be in P!

For this reason, Ladner's theorem can only work if we assume that $\mathsf{P} \neq \mathsf{NP}$.

Theorem 24.1 (Ladner's theorem)**.** *Assume that* $\mathsf{P} \neq \mathsf{NP}$*. Then there is a language* $A \in \mathsf{NP}$ *such that* $A \notin \mathsf{P}$ *and* A *is not* NP*-complete.*

The proof is a bit complicated. We will break up the proof this theorem into different sections, each describing a relevant idea. The last section will contain the final argument.

24.3.1 Padding language

To start the proof, we will start with a language C which is NP-complete, and then "pad" it to create an easier language A. We will define A to be

$$A = \{x01^{f(|x|)} : x \in C\}$$

for some function $f\colon \mathbb{N} \to \mathbb{N}$. That is, A contains all the strings in C, but padded with $f(n)$ ones (as well as a 0 before the string of 1s, so that we know where the string of 1s starts), where n is the length of the string that was in C.

We've seen padding before. The idea of it is that the padded language A is essentially equivalent to the original language C, but its inputs are longer; since we measure the amount of resources we use in terms of the input size, it will be "easier" to compute A than to compute C, because our time budget will be larger. For example, to show that $A \in \mathsf{P}$, we would merely need an algorithm that decides whether $x \in C$ in time that is polynomial in $|x01^{f(|x|)}| = |x| + 1 + f(|x|)$, rather than polynomial in $|x|$. If $f(|x|)$ is large, this is significantly easier than finding a polynomial-time algorithm for C.

The trick will be to try to choose f so that A has the desired properties. We will ensure that f is non-decreasing, and also that it is time-constructible, but the precise choice of the function f will have to wait a bit.

24.3.2 The padded language is in NP

We start by arguing that A is in NP. To do so, recall that we chose C to be in NP, so C has a polynomial-time verifier V. We now construct a polynomial-time verifier V' for A. On input $\langle y, w\rangle$, V' will check if y is of the form $y = x01^{f(|x|)}$ for some string x. Checking this requires computing $f(|x|)$, but that takes at most $O(f(|x|))$ time if f is time-constructible, so this is linear time in the input size. If the input is badly formatted, V' will reject.

If the input is of the right format, V' will then simulate $V(\langle x, w\rangle)$, and output what V outputs. Note that the string $\langle x, w\rangle$ is shorter than the input string $\langle y, w\rangle$ to V'; since V runs in polynomial time, it follows that V' also runs in polynomial time. Now, if $y \in A$, then $y = x01^{f(|x|)}$ for some string $x \in C$, and so some witness w of size polynomial in $|x|$ exists such that $V(\langle x, w\rangle)$ accepts; in this case, $V'(\langle y, w\rangle)$ will accept. Conversely, if $y \notin A$, then either y is not formatted like $x01^{f(|x|)}$ for any string x, or else it is formatted this way but x is not in C. Either way, there is no witness w that causes $V'(\langle y, w\rangle)$ to accept.

We conclude that V' is a polynomial-time verifier for A, so $A \in \mathsf{NP}$. This proof only required that $C \in \mathsf{NP}$ and that the function f was time-constructible.

24.3.3 If f is polynomial

What happens if we choose f to be a polynomial function, such as $3n^2$ or n^{10}?

In that case, we claim that $C \leq_m^p A$. To show this, we need a polynomial time mapping reduction from C to A. We describe a machine M to implement this reduction. Given input x, the machine M will compute $f(|x|)$ and then output $x01^{f(|x|)}$. Since f is time-constructible, this takes $O(f(|x|))$ time. Since we've chosen f to have polynomial growth rate, the running time of M is polynomial. Also, if $x \in C$ then $x01^{f(|x|)} \in A$ and if $x \notin C$ then $x01^{f(|x|)} \notin A$ by the definition of A. Hence M is a polynomial-time mapping reduction, so $C \leq_m^p A$.

We won't necessarily choose f to grow polynomially, but if we do (and if it's time-constructible) it will imply that $C \leq_m^p A$.

24.3.4 If f is super-polynomial

What happens if we choose f to grow super-polynomially? That is, $f(n) = \Omega(n^k)$ for each $k \in \mathbb{N}$?

In this case, we claim that if $C \leq_m^p A$ then C is in P. In other words, we claim that assuming C is not in P, we cannot have $C \leq_m^p A$. Since we are choosing C to be NP-complete and since we are assuming $\mathsf{P} \neq \mathsf{NP}$, we indeed get to assume that $C \notin \mathsf{P}$, so we will indeed conclude that there is no polynomial-time mapping reduction from C to A.

To see this, suppose we had a polynomial-time mapping reduction g from C to A. We first claim that we would also have a polynomial-time mapping reduction g' from C to A such that for each string y, $g'(y)$ is of the form $x01^{f(|x|)}$ for some string x. To see this, we will describe a Turing machine M computing g'. The first step of M on input y will be to compute $g(y)$ (which takes polynomial time in $|y|$). Next, M will check if $g(y)$ is formatted like $x01^{f(|x|)}$ for some string x. This requires computing $f(|x|)$, but f is time-constructible, so that takes $O(f(|x|))$, and the whole check can be done in linear time. If the formatting check passes, M will output $g(y)$. Otherwise, M will output $z01^{f(|z|)}$, where z is a fixed string not in C. The string z will be hard-coded into M, and printing out $z01^{f(|z|)}$ will take constant time as z is a fixed, constant string.

It is clear that M runs in polynomial time; indeed, it uses only a small overhead over the running time of the machine computing g. Moreover, if $y \in C$, then $g(y) \in A$, as g is a mapping reduction from C to A; but this means $g(y)$ is formatted like $x01^{f(|x|)}$, so $M(y)$ outputs $g(y)$, which is in A. On the other hand, if $y \notin C$, then $g(y) \notin A$, and there are two options: either $g(y)$ is badly formatted, in which case $M(y)$ outputs $z01^{f(|z|)}$, or else it is correctly formatted, in which case $M(y)$ outputs $g(y)$ and $g(y) = x01^{f(|x|)}$ for some string x not in C. Either way, the output of $M(y)$ looks like $x01^{f(|x|)}$ for some string x which is not in C. This confirms that M computes a polynomial-time mapping reduction g' that gives strings of the desired format.

Let h denote the function that maps y to the string x for which $g'(y) = x01^{f(|x|)}$. Then h is computable in polynomial time, and for all strings y, we have $y \in C$ if and only if $h(y) \in C$.

Next, note that since g' is computable in polynomial time in $|y|$, the size $|g'(y)|$ must grow polynomially with $|y|$. Hence, for sufficiently large y, we have $f(|y|) > |g'(y)|$ (since f grows faster than any polynomial). This means that there is some number $n_0 \in \mathbb{N}$ such that for all strings y with $|y| \geq n_0$, we have $|g'(y)| < f(|y|)$. Since $g'(y) = x01^{f(|x|)}$ for some x, we must have $|x|+1+f(|x|) < f(|y|)$ whenever $|y| \geq n_0$. This implies that when $|y| \geq n_0$, we have $f(|x|) < f(|y|)$, and since f is non-decreasing, $|x| < |y|$. In other words, for all strings y with $|y| \geq n_0$, we have $|h(y)| < |y|$, and $h(y) \in C$ if and only if $y \in C$.

Now consider the Turing machine which takes input y and applies h to y repeatedly, for a total of $|y|$ times. Since each application of h results in a shorter string if $|y| \geq n_0$, or else results in a string of length at most n_0 if $|y| < n_0$, we conclude that each application of h can be computed in polynomial time in $|y|$ (since n_0 is a constant). Moreover, we are doing only $|y|$ such computations. Hence this Turing machine runs in polynomial time in the input size $|y|$. Let h' be the function computed by this Turing machine.

Then for every string y, we have $h'(y) \in C$ if and only if $y \in C$; moreover, since each application of h decreases the size of y by 1 unless $|y| < n_0$, it must be the case that $|h'(y)| \leq n_0$ for all y. Therefore, in polynomial time, we can compute a function $h'(y)$ which maps y to a string of length at most n_0 such that $h'(y)$ is in C if and only if $y \in C$. In other words, h' is a polynomial-time mapping reduction from C to $C \cap \{y : |y| < n_0\}$. But the latter is a finite set! Every finite set is in P, because we can create a Turing machine that has all the answers hard-coded using the internal states. So we have a polynomial-time mapping reduction from C to a language in P, and hence $C \in \mathsf{P}$, as desired.

This proof only required that f is non-decreasing, time-constructible, and grows faster than any

polynomial.

24.3.5 The desired choice of f

Let's assume we chose C to be NP-complete, and let's also assume that $\mathsf{P} \neq \mathsf{NP}$. Then we saw that if we choose f to grow polynomially, we will have $C \leq_m^p A$, and if we choose f to grow super-polynomially, we will not have $C \leq_m^p A$.

What we want to do now is to choose f so that it grows polynomially if $A \in \mathsf{P}$ and super-polynomially if $A \notin \mathsf{P}$. This way, if $A \in \mathsf{P}$ then we will have $C \leq_m^p A$, implying that $C \in \mathsf{P}$, giving a contradiction (since we've assumed that C is NP-complete and that $\mathsf{P} \neq \mathsf{NP}$). This will therefore mean that $A \notin \mathsf{P}$, but that would mean that f grows super-polynomially, and so there is no polynomial-time mapping reduction from C to A, and since $C \in \mathsf{NP}$ that implies that A is not NP-hard. This will force A to be NP-intermediate.

So we want f to grow polynomially if $A \in \mathsf{P}$ and super-polynomially if $A \notin \mathsf{P}$; if we could arrange this (and if f were non-decreasing and time constructible), we would be done. The problem is that the definition of A depends on f, so this seems circular: the choice of f depends on A, and the definition of A depends on f. To handle this, we will do something that essentially amounts to a diagonalization argument.

24.3.6 The diagonalization argument

Recall that we want f to grow polynomially if $A \in \mathsf{P}$ and super-polynomially if $A \notin \mathsf{P}$; recall also that

$$A = \{x01^{f(|x|)} : x \in C\},$$

and that f needs to be non-decreasing and time-constructible.

We will construct f by describing a Turing machine M_f computing it. To be time-constructible, we want $M_f(0^n)$ to output $\langle f(n) \rangle$ in $O(f(n))$ time. Actually, we will define $f(n)$ inductively, so that $f(n)$ depends on the values of $f(k)$ for $k < n$. We start by setting some base cases such as $f(0) = 0$ and $f(1) = 1$.

Next, on input 0^n, the machine M_f will go through the first n Turing machines $M_1, M_2, \ldots, M_n$ in order (for example, in lexicographic order of their encodings as strings). For each machine M_i, M_f will attempt to disprove the claim "M_i decides A in polynomial time". (We will soon discuss how M_f will do this.) Then M_f will keep track of the smallest i for which this claim was not disproven, and it will output $\langle n^{i+3} \rangle$. The idea is that if some Turing machine M decides A in polynomial time, then for some integer i we will have $M_i = M$, and hence for all $n \geq i$, the machine M_f will fail to disprove that M_i decides A, meaning that $M_f(0^n)$ will return $\langle n^{i+3} \rangle$ for all sufficiently large n; this will cause f to grow polynomially if A is in P. On the other hand, if no Turing machine decides A in polynomial time, then eventually, M_f should disprove the claim "M_i decides A in polynomial time" for every given i, and hence f will grow faster than n^{i+3} for every $i \in \mathbb{N}$; this will cause f to grow super-polynomially if A is not in P.

Before we describe M_f in more detail, we will introduce the machine M_C which decides C in exponential time. This machine exists since $\mathsf{NP} \subseteq \mathsf{EXP}$, which means that since $C \in \mathsf{NP}$ we have $C \in \mathsf{EXP}$. In particular, we have $C \in \mathsf{TIME}(2^{n^k})$ for some $k \in \mathbb{N}$. We let M_C be a Turing machine which decides C and has running time $O(2^{n^k})$.

OK, back to M_f. On input 0^n, its first step will be to calculate $\ell = (\log n)^{1/(k+1)}$. This number ℓ is sufficiently small that simulating M_C on an input string of size at most ℓ can be done in $O(n)$ time. This calculation of ℓ can actually be done in $O(n)$ time (first M_f calculates $\langle n \rangle$ in binary from its input 0^n, and then it can perform computations on $\langle n \rangle$ that are even allowed to take exponential

time, and these will still take only $O(n)$ time because the "input size" for them is $|\langle n \rangle| = O(\log n)$). Note that there are also only at most $O(n)$ strings of length at most ℓ.

Next, M_f will write down the values of $f(k)$ for all $k \leq \ell$ by recursively calling itself on smaller inputs. We will soon show that M_f will take only $O(n^3)$ time for all its calculations other than this recursive step; but using dynamic programming (i.e. by computing $f(0)$, then $f(1)$, then $f(2)$, etc., and looking these up in memory instead of making recursive calls), we can making only ℓ total calls to M_f on such inputs $k \leq \ell$, each with input 0^ℓ or less. This means the total time to compute all the values of $f(k)$ for $k \leq \ell$ will be $O(\ell^4)$, which is much smaller than $O(n)$.

The next step of M_f will be to iterate over all strings s of length at most ℓ; recall that there are at most $O(n)$ such strings. For each such string s, M_f will decide whether s is in A. To do so, M_f will need to check if s looks like $x01^{f(|x|)}$ for some string x; luckily, M_f already wrote down the values of $f(k)$ for all $k \leq \ell$, so this check is easy and can be done in linear time in ℓ. If the formatting check passes, M_f will also need to check whether x is in C, which it can do by simulating $M_C(x)$; recall that this can be done in $O(n)$ steps. Therefore, after $O(n^2)$ total steps, M_f will produce a table that lists, for each string s of length at most ℓ, whether s is in A.

The next step of M_f will be to iterate over the first n Turing machines $M_1, M_2, \ldots, M_n$ in order. It takes only $O(n)$ time to list them. For each machine, M will again iterate over all strings of length at most ℓ. Then for each pair of machine M_i and string s with $|s| \leq \ell$, the machine M_f will simulate $M_i(s)$ for n steps. If $M_i(s)$ halts in that time, M_f will check whether it correctly decides if $s \in A$ (by consulting the table). M_f will then find the smallest number j such that some machine M_i for $i \leq j$ correctly decided if $s \in A$ for all strings s of length at most ℓ, and the running time of M_i on each string s was at most $|s|^j + j$. If there is no such j, M_f will find the smallest j for which this property wasn't properly checked (due to the time constraint of n steps). Then M_f will output n^{j+3}.

Since there are n machines and at most $O(n)$ strings, all the simulations together will take $O(n^3)$ steps, and hence M_f runs in $O(n^3)$ time. Additionally, it should be clear that f is increasing, because the value of j that M_f finds can only increase. It is also clear that $f(n) = \Omega(n^3)$, and hence f is time-constructible, since M_f outputs $f(n)$ in at most $O(f(n))$ time.

Now, if $A \in \mathsf{P}$, it means there is some Turing machine M_i that decides A in polynomial time. The running time of M_i polynomial, so for sufficiently large j, it takes fewer than $|s|^j + j$ steps to halt on each string s. Since M_i decides A, it therefore follows that for all sufficiently large $n \in \mathbb{N}$, we will have $f(n) = n^{\max\{i,j\}+3}$. Hence f grows polynomially if $A \in \mathsf{P}$, and as we've seen, that causes $C \leq_m^p A$ to hold, so $C \in \mathsf{P}$, which gives a contradiction. From this we conclude that $A \notin \mathsf{P}$.

Next, if $A \notin \mathsf{P}$, then for every machine M_i and every $j \in \mathbb{N}$, there will be some string s such that either $M_i(s)$ gives the wrong answer as to whether $s \in A$, or else $M_i(s)$ takes more than $|s|^j + j$ steps. This means that there will not be any $j \in \mathbb{N}$ such that $f(n) \leq n^j + j$ for all $n \in \mathbb{N}$. From this it follows that $f(n)$ cannot be upper bounded by any polynomial function, since every polynomial can be upper bounded by a function of the form $n^j + j$. In other words, $f(n)$ grows super-polynomially, and as we've seen, this implies that there is no polynomial-time mapping reduction from C to A. Since $C \in \mathsf{NP}$, it follows that A is not NP-hard, so it is NP-intermediate, as desired.

24.4 Final remarks

We've now covered the basics of some complexity classes, such as P, NP, and PSPACE. The class P can be thought of as containing the efficiently solvable problems. Problems in NP are essentially puzzles, and have efficiently-checkable solutions. Problems that are NP-hard are as hard to solve as any puzzle: if you can solve such a problem, then you can solve any problem in NP with only

polynomial overhead. Since we generally don't know how to show that a given language is not in NP, the usual way of showing that a language is intractably hard is to show that it is NP-hard (then the language is not in P assuming $\mathsf{P} \neq \mathsf{NP}$).

Finally, Ladner's theorem tells us that we cannot hope for a clean characterization in which every single language is either in NP or else is NP-hard; there will inevitably be languages that are NP-intermediate, assuming that $\mathsf{P} \neq \mathsf{NP}$. So while in practice, people argue that languages are hard by showing they are NP-hard, we know that this cannot always work, and there are some languages that do not have efficient algorithms but which cannot be shown to be NP-hard (at least, assuming that $\mathsf{P} \neq \mathsf{NP}$).