15.1 Proof of the Cook-Levin Theorem: SAT is NP-complete

- Already know $\mathrm{SAT} \in \mathrm{NP}$, so only need to show SAT is NP-hard.
- Let L be any language in NP. Let M be a NTM that decides L in time n^k.

  We define a polynomial-time reduction

  $$f_L : \text{inputs} \mapsto \text{formulas}$$

  such that for every w,

  $$M \text{ accepts input } w \text{ iff } f_L(w) \text{ is satisfiable}$$

Reduction via "computation histories"

Proof Idea: satisfying assignments of $f_L(w) \leftrightarrow$ accepting computations of M on w

Describe computations of M by boolean variables:

- If $n = |w|$, then any computation of M on w has at most n^k configurations.
- Each configuration is an element of C^{n^k}, where $C = Q \cup \Gamma \cup \{\#\}$ (mark left and right ends with $\{\#\}$).

$\rightsquigarrow$ computation depicted by $n^k \times n^k$ "tableau" of members of C.

- Represent contents of cell (i,j) by $|C|$ boolean variables $\{x_{i,j,s} : s \in C\}$, where $x_{i,j,s} = 1$ means "cell (i,j) contains s".
- $0 \le i,j < n^k$, so $|C| \cdot n^{2k}$ boolean variables in all

Subformulas that verify the computation

Express conditions for an accepting computation on w by boolean formulas:

- $\phi_{\text{cell}} =$ "for each (i,j), there is exactly one $s \in C$ such that $x_{i,j,s} = 1$".

  We can express this by

$$\phi_{\text{cell}} = \bigwedge_{0 \le i,j < n^k} \left(\left(\bigvee_{s \in C} x_{i,j,s} \right) \wedge \left(\bigwedge_{\substack{s,t \in C \\ s \ne t}} (\bar{x}_{i,j,s} \vee \bar{x}_{i,j,t}) \right) \right)$$

The "or" over $s \in C$ ensures that cell (i, j) has at least one symbol written in it, and the "or" over $s \neq t \in C$ ensures that at most one symbol is written in the cell. Thus, overall, we ensure that *exactly* one symbol is written in each cell.

- ϕ_{start} = "first row equals start configuration on w"

$$\phi_{\text{start}} = x_{0,0,\#} \wedge x_{0,1,q_0} \wedge \left(\bigwedge_{j=1}^{n} x_{0,j+1,w_j} \right) \wedge \left(\bigwedge_{j=n+2}^{n^k-2} x_{0,j,\sqcup} \right) \wedge x_{0,n^k-1,\#}$$

- ϕ_{accept} = "this computation branch accepts w" (we will look for q_{accept} being *somewhere* in the table)

$$\phi_{\text{accept}} = \bigvee_{0 \le i,j < n^k} x_{i,j,q_{accept}}$$

- ϕ_{move} = "every 2×3 window is consistent with the transition function of M"

$$\phi_{\text{move}} = \bigwedge_{\substack{0<i<n^k-1 \\ 0 \le j < n^k-1}} (\text{the } 2 \times 3 \text{ window with top-middle cell at } (i, j) \text{ is valid})$$

To know whether such a cell is "valid", we generate a set V of 6-tuples of assignments to cells in a 2×3 window that are valid. We do not delve into the tedious details here of how to define V, but you may try as an exercise (for example, '#' should be copied to the row below without change, the top-middle cell, if not a state or adjacent to a state, should be copied to the cell below without change, etc.). Then the expression "the 2×3 window with top-middle cell at (i, j) is valid" can then be expressed as

$$\bigvee_{(a_1,\ldots,a_6) \in V} (x_{i,j-1,a_1} \wedge x_{i,j,a_2} \wedge x_{i,j+1,a_3} \wedge x_{i+1,j-1,a_4} \wedge x_{i+1,j,a_5} \wedge x_{i+1,j+1,a_6})$$

Completing the proof

Claim: Each of above can be expressed by a formula of size of size $O((n^k)^2) = O(n^{2k})$, and can be constructed in polynomial time from w.

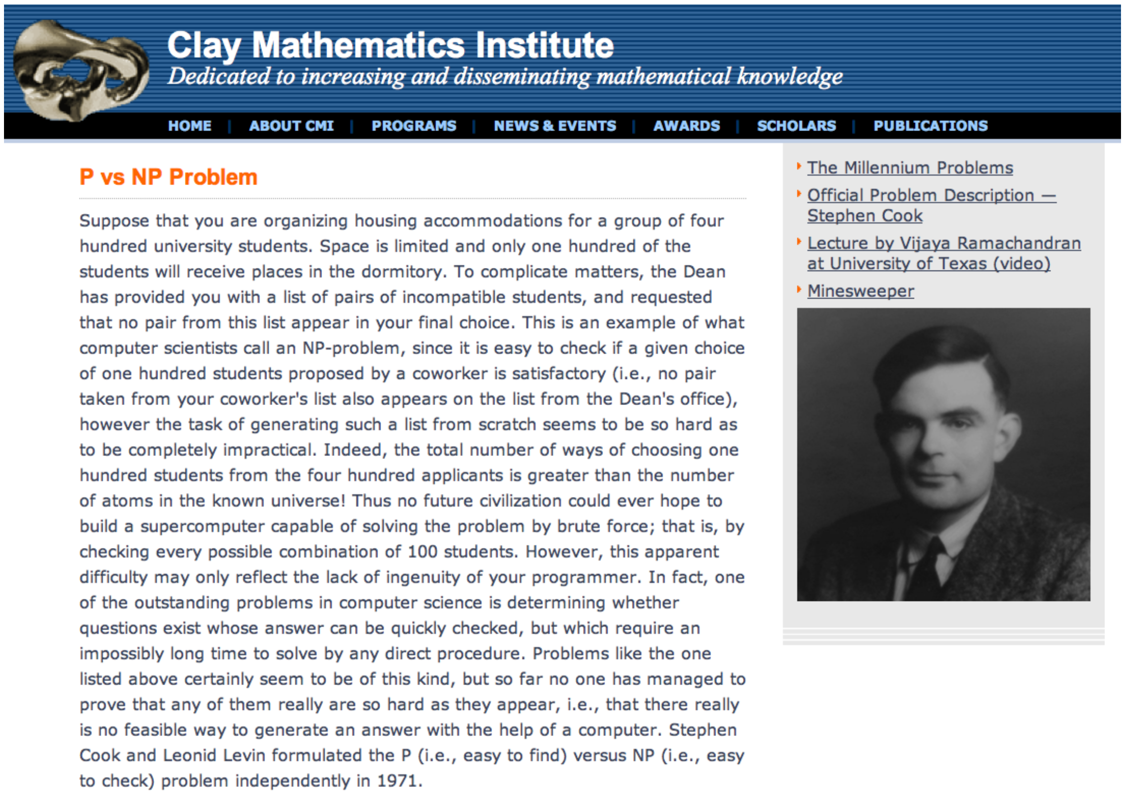
Claim: M has an accepting computation on w if and only if $f_L(w) = \phi_{\text{cell}} \wedge \phi_{\text{start}} \wedge \phi_{\text{accept}} \wedge \phi_{\text{move}}$ has a satisfying assignment.

Pf: ϕ_{cell} ensures each cell has exactly one symbol written in it, so that we have a valid tableau. ϕ_{accept} makes sure we arrive in an accept state, and ϕ_{start} ensures the first row of the tableau is valid. Now, we claim by induction that the first j rows of the tableau represent a valid execution branch of the nondeterministic Turing machine for each j, by induction on j. The base case is true for $j = 1$ due to ϕ_{start}. Now for the inductive step. For $j > 1$, we can assume the $(j-1)$st row is valid. Now, for each cell in the $(j-1)$st row which contains a non-#, non-state symbol and which is not adjacent to a state symbol, it must be the top-center cell in *some* 2×3 window. The portion of ϕ_{move} corresponding to that window ensures that the cell is copied without change to row j. Also for the cell with the state location in row $j-1$, it also is the top-center cell in some 2×3 window, and that window ensures the transition function is faithfully represented in the tableau when moving from row $j-1$ to j.

Thus $w \mapsto f_L(w)$ is a polynomial-time reduction from L to SAT.

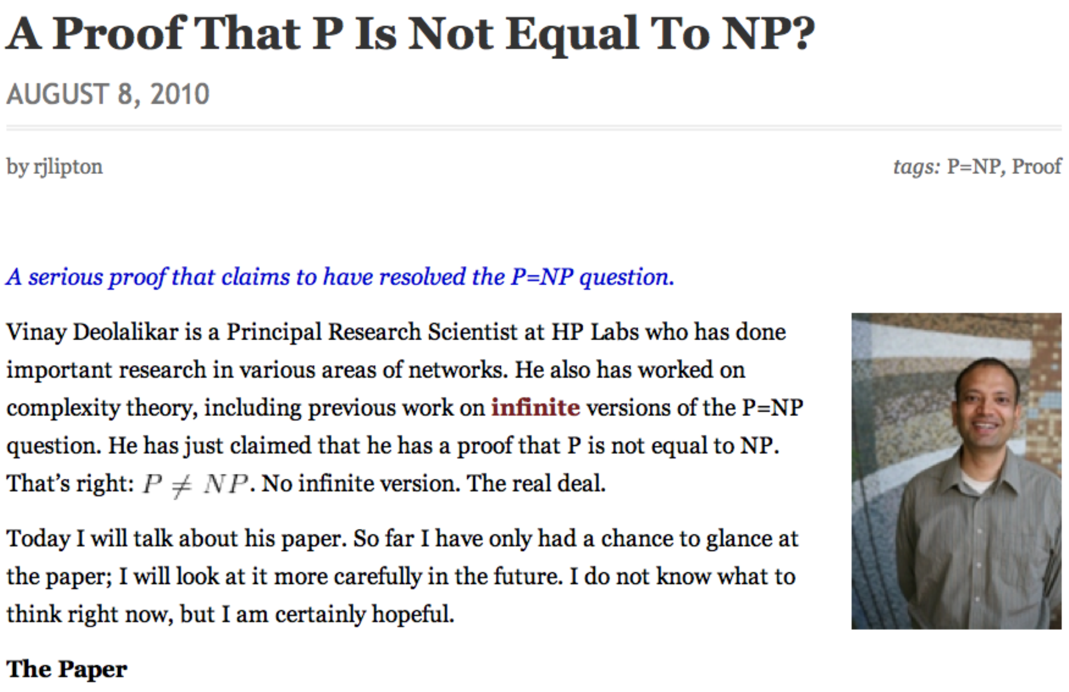
Since above holds for every $L \in \text{NP}$, SAT is NP-hard, as desired. ■

15.2 Towards Resolving P vs. NP


Clay Mathematics Institute
Dedicated to increasing and disseminating mathematical knowledge
HOME | ABOUT CMI | PROGRAMS | NEWS & EVENTS | AWARDS | SCHOLARS | PUBLICATIONS
The Millennium Problems
Official Problem Description — Stephen Cook
Lecture by Vijaya Ramachandran at University of Texas (video)
Minesweeper

P vs NP Problem

Suppose that you are organizing housing accommodations for a group of four hundred university students. Space is limited and only one hundred of the students will receive places in the dormitory. To complicate matters, the Dean has provided you with a list of pairs of incompatible students, and requested that no pair from this list appear in your final choice. This is an example of what computer scientists call an NP-problem, since it is easy to check if a given choice of one hundred students proposed by a coworker is satisfactory (i.e., no pair taken from your coworker's list also appears on the list from the Dean's office), however the task of generating such a list from scratch seems to be so hard as to be completely impractical. Indeed, the total number of ways of choosing one hundred students from the four hundred applicants is greater than the number of atoms in the known universe! Thus no future civilization could ever hope to build a supercomputer capable of solving the problem by brute force; that is, by checking every possible combination of 100 students. However, this apparent difficulty may only reflect the lack of ingenuity of your programmer. In fact, one of the outstanding problems in computer science is determining whether questions exist whose answer can be quickly checked, but which require an impossibly long time to solve by any direct procedure. Problems like the one listed above certainly seem to be of this kind, but so far no one has managed to prove that any of them really are so hard as they appear, i.e., that there really is no feasible way to generate an answer with the help of a computer. Stephen Cook and Leonid Levin formulated the P (i.e., easy to find) versus NP (i.e., easy to check) problem independently in 1971.



A Proof That P Is Not Equal To NP?

AUGUST 8, 2010

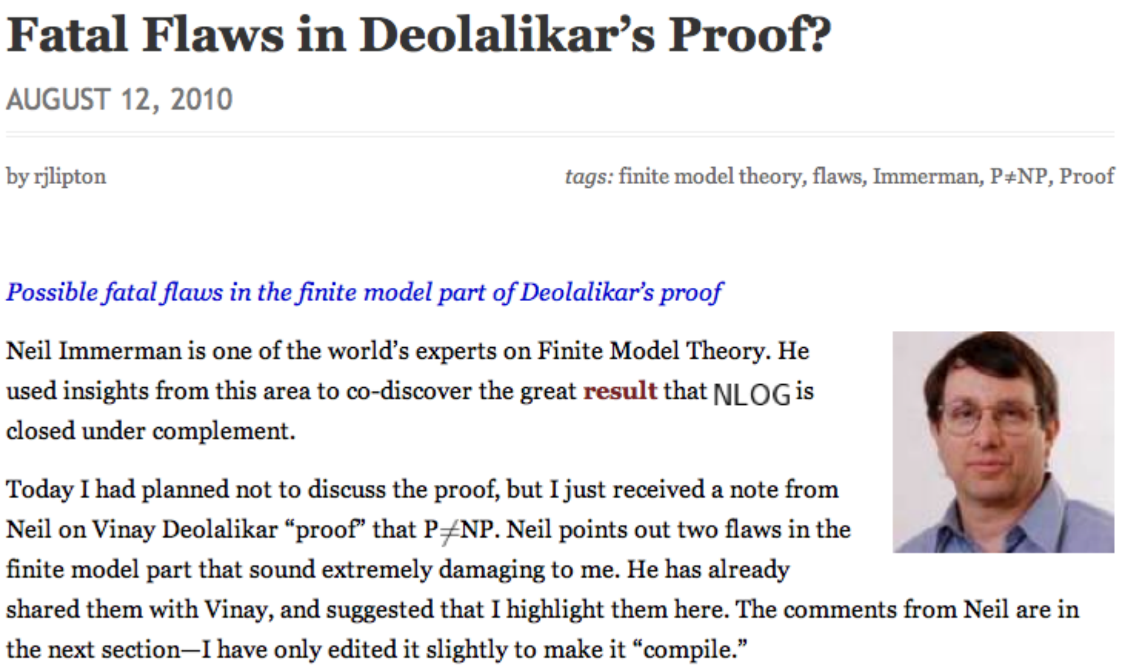
by rjlipton *tags:* P=NP, Proof

A serious proof that claims to have resolved the P=NP question.

Vinay Deolalikar is a Principal Research Scientist at HP Labs who has done important research in various areas of networks. He also has worked on complexity theory, including previous work on **infinite** versions of the P=NP question. He has just claimed that he has a proof that P is not equal to NP. That's right: $P \neq NP$. No infinite version. The real deal.

Today I will talk about his paper. So far I have only had a chance to glance at the paper; I will look at it more carefully in the future. I do not know what to think right now, but I am certainly hopeful.

The Paper



Fatal Flaws in Deolalikar's Proof?

AUGUST 12, 2010

by rjlipton *tags:* finite model theory, flaws, Immerman, P≠NP, Proof

Possible fatal flaws in the finite model part of Deolalikar's proof

Neil Immerman is one of the world's experts on Finite Model Theory. He used insights from this area to co-discover the great **result** that NLOG is closed under complement.

Today I had planned not to discuss the proof, but I just received a note from Neil on Vinay Deolalikar "proof" that P≠NP. Neil points out two flaws in the finite model part that sound extremely damaging to me. He has already shared them with Vinay, and suggested that I highlight them here. The comments from Neil are in the next section—I have only edited it slightly to make it "compile."

15.3 Around and Within NP

co-NP

co-NP $= \{\overline{L} : L \in \text{NP}\}$.

Some co-NP-complete problems:

- Complement of any NP-complete problem.
- TAUTOLOGY $= \{\varphi : \forall a\ \varphi(a) = 1\}$ (even for 3-DNF formulas φ).

Believed that NP $\neq$ co-NP, P $\neq$ NP $\cap$ co-NP.

Between P and NP-complete

We will prove the following theorem at the end of the notes.

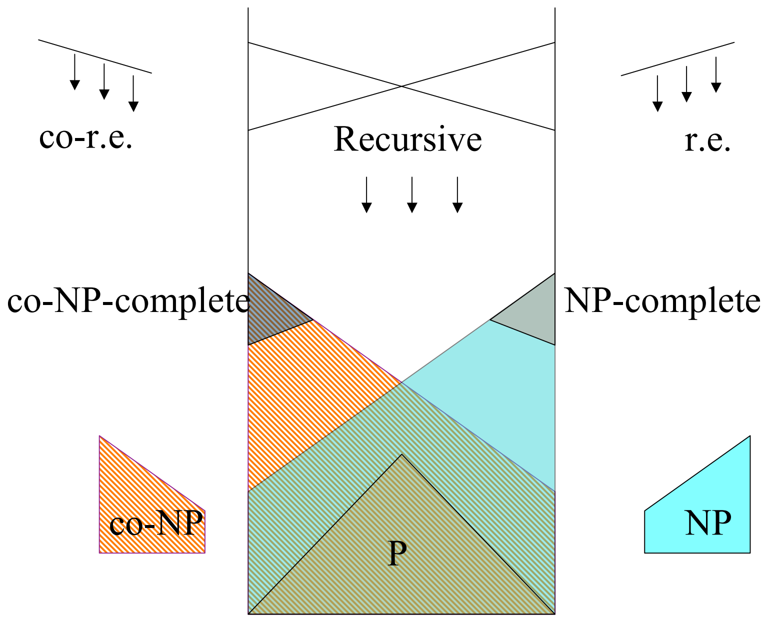
Theorem: If P $\neq$ NP, then there are NP languages that are neither in P nor NP-complete.

Some natural candidates:

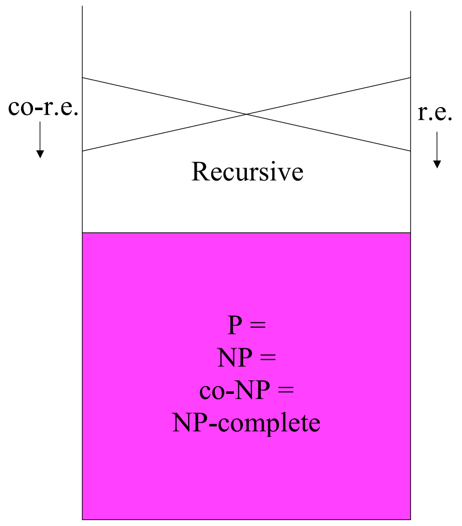
- FACTORING (when described as a language)
- NASH EQUILIBRIUM
- GRAPH ISOMORPHISM
- Any problem in NP $\cap$ co-NP for which we don't know a poly-time algorithm.

15.4 Two Possible Worlds

The World If P $\neq$ NP



The World If P = NP



15.5 NP-intermediate problems

Another interesting concept is that of *NP-intermediate* problems. The question here is: if **P** $\neq$ **NP**, can it be the case that every $L \in$ **NP** is *either* in **P**, *or* is **NP**-complete? Or must there necesarily be some languages which are in between? (i.e. in **NP**, but too hard to be in **P** and too easy to be **NP**-hard)

As mentioned above, there must be some **NP**-intermediate languages. This is known as *Ladner's theorem* (as it was proven by Richard Ladner).

We will prove the following in the next section.

Lemma 15.1 *There is an infinite enumeration of Turing machines $M_1, M_2, \ldots$, such that:*

(1) $\mathbf{P} = \{L : \exists i,\ L = L(M_i)\}$

(2) M_i runs in time at most $|x|^i$ for each i, for any x with $|x| \geq 2$

(3) There is an algorithm which, given i as input, outputs the encoding $\langle M_i \rangle$ in time at most polylogarithmic in i

Similarly, there is an enumeration of all poly-time computable functions $f_1, f_2, \ldots$ such that f_j runs in time at most $|x|^j$ for any input x with $|x| \geq 2$, and there is an algorithm which given i as input, outputs the encoding of a Turing machine computing f_i in time polylogarithmic in i.

Now we prove Ladner's theorem.

Theorem 15.2 *If* $\mathbf{P} \neq \mathbf{NP}$*, then there exists a language* $L \in \mathbf{NP} \backslash \mathbf{P}$ *such that there is no poly-time Karp reduction from* SAT *to L.*

Proof: Let $M_1, M_2, \ldots$ and $f_1, f_2, \ldots$ be sequences satisfying the conditions of Lemma 15.1.

The remainder of the following proof is now due to Impagliazzo (see the pdf at the url `http://oldblog.computationalcomplexity.org/media/ladner.pdf`). We now define a language L which we show is **NP**-intermediate. Define $L = \{\langle \phi \rangle 1^{f(n)-n} : |\langle \phi \rangle| = n,\ \phi \in \mathrm{SAT}\}$, where $\langle \phi \rangle$ is the encoding of some Boolean formula ϕ such that the encoding ends in a 0. It remains to define f to ensure that L is **NP**-intermediate. In order to accomplish this, we must make sure that (a) $L \neq L(M_i)$ for any M_i in the above sequence, and (b) none of the f_i are valid Karp-reductions from SAT to L. We will also ensure that f is poly-time computable. Note we cannot make $f(n)$ too large, since otherwise L would be in **P** (e.g. $f(n) \geq 2^n$). We cannot make $f(n)$ at most a polynomial in n either though, since otherwise there's a clear poly-time Karp reduction from SAT to L (append $f(n)$ 1's).

We define $f(n)$ via an algorithm. Initialize $i = 1$. Then for $1 \leq m \leq n$ in order, first define $f(m) = m^i$. Then search all x with $2 \leq |x| \leq \lg m$. If any such x exists with $x \in L(M_i)$ but $x \notin L$, or vice versa, then we set $i \leftarrow \min\{i+1, \lg m / \lg\lg m\}$ (we have successfully demonstrated that $L \neq L(M_i)$, so we increment i as long as it isn't "too big" already, to keep our computation of $f(n)$ efficient). Note $f(n)$ is poly-time computable in n since there are $2^{\lg n} = n$ x's of such small length to check, and i is never larger than $\lg n / \lg\lg n$ so we can run M_i on x in time at most $|x|^{\lg n / \lg\lg n} = n$ to check whether $x \in L(M_i)$. We can also check whether $x \in L$ in poly-time since the formula portion of x has length at most $\lg n$, and SAT can be decided in exponential time, and $2^{O(\lg n)} \leq poly(n)$.

We claim that now, with this definition of $f(n)$ in hand, L is in $\mathbf{NP} \backslash \mathbf{P}$. To see that it is in **NP**, note we have to check that the longest suffix of 1's is of length $f(n) - n$, which is easy to check since f is poly-time computable. We also need to check that $\langle \phi \rangle \in \mathrm{SAT}$. But this can be checked with a poly-time verifiable witness: namely the SAT witness (i.e. a satisfying assignment for ϕ). Now we must show that $L \notin \mathbf{P}$. If $L \in \mathbf{P}$, then there exists some j such that $L = L(M_j)$. But then, once the variable i in the algorithm to compute f equals j, i can never increase beyond

j, since there will never be an x of *any* length which is in exactly one of L or $L(M_j)$ but not both. This implies that $f(n) = O(n^j)$ for all n. But then there is an easy poly-time algorithm to decide SAT: given a Boolean formula ϕ, append $f(n) - |\langle\phi\rangle|$ 1's to $\langle\phi\rangle$ to form a string x, then run M_j on x. The runtime of this procedure is at most the runtime of M_j plus $O(n^j)$, which overall is polynomial in n.

We now just need to ensure that no f_j is a valid Karp-reduction from SAT to L. Suppose, to the contrary, that some f_j is a valid Karp-reduction from SAT to L. We now show this implies that SAT $\in$ **P**, which is a contradiction. Let ϕ be a formula for which we wish to decide whether $\phi \in$ SAT. We now describe how our poly-time algorithm $\mathcal{A}$ operates to decide whether $\phi \in$ SAT. We have $|f_j(\phi)| \le |\langle\phi\rangle|^j$. Since we have established $L \notin \mathbf{P}$, there must be some constant n_0 such that $f(n) \ge n^k$ for all $n \ge n_0$, where $k > j$. We hardcode the answer to all ϕ with $|\langle\phi\rangle| \le n_0$. Thus $\mathcal{A}$ can easily output the correct answer if $|\langle\phi\rangle| \le n_0$.

If $|\langle\phi\rangle| > n_0$, we compute $f_j(\phi)$. If the output is not of the correct format for L (i.e. the encoding of a boolean formula ψ followed by $f(|\langle\psi\rangle|)$ 1's, then we reject. Otherwise, let $m = |\langle\psi\rangle|$. If $m \le n_0$, then we again can output the answer from memory since it is hardcoded. Otherwise, we have $m^k \le f(m) = |f_j(\phi)| \le |\langle\phi\rangle|^j$, implying $m = |\langle\psi\rangle| \le |\langle\phi\rangle|^{j/k}$ for $f(m) = m^k$. If $k \le j$ then we must have $|\langle\psi\rangle| \le n_0$, so we can decide SAT for ψ (and hence for ϕ) in constant time by looking up our hardcoded answer. Otherwise, $|\langle\psi\rangle| \le |\langle\phi\rangle|^{j/k} < |\langle\phi\rangle|$, and hence ψ is a strictly smaller formula than ϕ. We then apply this algorithm recursively to decide $\psi \in SAT$, which leads to a poly-time algorithm. ■

15.6 Proof of Lemma 15.1

As we will see in the next lecture, the set of all Turing machines is countable. Thus the set of all decidable languages is countable, implying that **P** as a set of languages is also countable. This implies there is some enumeration of Turing Machines $M'_1, M'_2, \ldots$ such that $\mathbf{P} = \{L : \exists i,\ L = L(M'_i)\}$. The issue is that (a) it is not clear how to efficiently iterate over the M'_i, and (b) M'_i does not necessarily run in time at most n^i.

Let us now fix the above issues. For (1), we proceed by treating *every* binary string with a single # symbol as the encoding of a pair $\langle M, k\rangle$ (the # separates the encoding of M from the binary description of k). Here k is a positive integer, and M is some Turing machine. If some binary string does not encode a valid TM M, then we treat it as an encoding of the Turing machine M^* which always immediately transitions into q_{reject} in the very first step. Otherwise, if it does encode such a pair, we interpret it as the Turing machine which simulates M on its input x, $|x| = n$, but halts if M takes more than n^k steps. This simulation is accomplished by a Turing machine having four tapes. In the beginning of the computation, the first tape is used to compute n^k and store it, taking $O(k^3 \log^2 n)$ time (k repeated multiplications of integers that are $O(k \log n)$ digits). The second tape is used to keep track of t, the number of steps taken by the simulator so far. The third tape is used to store $\langle M\rangle$, and the the input x is copied to the final work tape (the fourth tape). After every step of the simulator, t is incremented, taking $O(k \log n)$ time. We also test whether $t > n^k$ after every step of the simulator, and if so, we halt and reject. If M halts before that, we halt in the state M halts in. This simulation takes time at most $O(k^3 \log^2 n + n^k) \le Cn^k \le n^{k'}$ for $n \ge 2$, where $k' = k + \lceil \log_2 C \rceil$.

Our final enumeration is thus as follows: $M_1 = M_2 = \ldots = M_{\lceil \log_2 C\rceil} = M^*$. From then on, we enumerate over (i,k) pairs in the order $(1,1), (1,2), (2,1), (1,3), (2,2), (3,1), \ldots$ (first all pairs summing to two, then those summing to three, etc.). For each (i,k) pair, we treat i as a description of M (which again may map to M^* is i written in binary is not a valid encoding of some TM), and k is as described above. By construction, each TM in this sequence decides a language in **P**. Furthermore, for each $L \in \mathbf{P}$, there is *some* TM M (corresponding to an integer i when encoded in binary) deciding it in time at most n^k for some $k \ge 1$ for all $n \ge 2$. Since the simulation above can blow up the running time by at most some fixed polynomial factor (say n^j for some j), this means when the pair $(i, k+j)$ is enumerated over, M will always be faithfully executed on its input without being cut off early.