Chapter 6: The Deep Question: **P** vs. **NP**

ECS 220: Theory of Computation

based on “The Nature of Computation” by Moore and Mertens

The deep question: **P** vs. **NP**

The evidence in favor of Cook's and Valiant's hypotheses is so overwhelming, and the consequences of their failure are so grotesque, that their status may perhaps be compared to that of physical laws rather than that of ordinary mathematical conjectures.

–Volker Strassen

(**P vs. NP**) vs. other major open mathematical questions

Clay Millenium Prize problems

- Riemann hypothesis: *zeros of* $\zeta(s) = \sum_{n=1}^{\infty} \frac{1}{n^s}$
- Poincaré conjecture: *high-dimensional spheres*
- Navier-Stokes equations: *turbulent hydrodynamic flows*
- Yang-Mills existence and mass gap: *high-energy physics*
- Birch and Swinnerton-Dyer Conjecture: *elliptic curves/number theory*
- Hodge Conjecture: *algebraic topology*
- **P** vs. **NP**

The others are important problems, but **P** vs. **NP** is about *the nature of problem-solving itself*.

Is it harder to find solutions than to check them?

Grotesque consequence #1: The Great Collapse

- If **P=NP**, then many other classes also equal **P**
  e.g., if **P=NP**, then **P=coNP** (why?)
- Recall: (below, B is a predicate in **P**, and $|w| = \text{poly}(|x|)$)
  NP = predicates of the form $A(x) = (\exists w)\ B(x,w)$
  coNP = predicates of the form $A(x) = (\forall w)\ B(x,w)$
  $\mathbf{\Sigma_2 P}$ = predicates of the form $A(x) = (\exists y)(\forall z)\ B(x,y,z)$

SMALLER-BOOLEAN-CIRCUIT $\in \mathbf{\Sigma_2 P}$

Given: Boolean circuit C computing predicate φ_C: $\{0,1\}^n \rightarrow \{0,1\}$

Question: Is there a smaller circuit computing φ_C?

answer = yes $\Leftrightarrow$ $(\exists$ circuit $D)\ |D| < |C|$ and $(\forall$ inputs $w)\ C(w) = D(w)$

The polynomial hierarchy

SMALL**ER**-BOOLEAN-CIRCUIT $\in \mathbf{\Sigma_2 P}$
Given: Boolean circuit C
Question: Is there a smaller circuit computing φ_C?

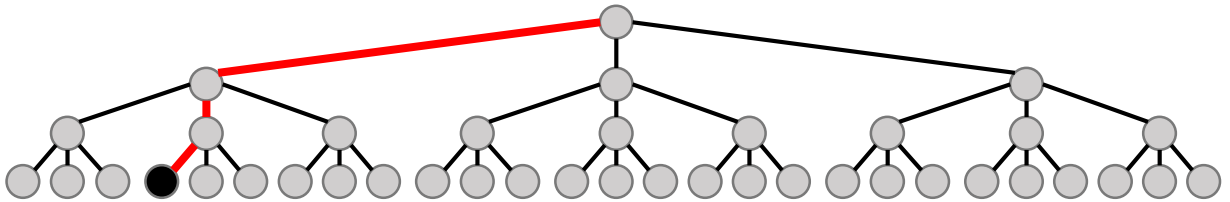
SMALL**EST**-BOOLEAN-CIRCUIT $\in \mathbf{\Pi_2 P}$
Given: Boolean circuit C
Question: Is C the smallest circuit computing φ_C?

$\mathbf{NP} = \mathbf{\Sigma_1 P}$: $A(x) = (\exists w)\ B(x,w)$
$\mathbf{coNP} = \mathbf{\Pi_1 P}$: $A(x) = (\forall w)\ B(x,w)$
$\mathbf{\Sigma_2 P}$: $A(x) = (\exists w_1)(\forall w_2)\ B(x,w_1,w_2)$
$\mathbf{\Pi_2 P} = \mathbf{co\Sigma_2 P}$: $A(x) = (\forall w_1)(\exists w_2)\ B(x,w_1,w_2)$
$\mathbf{\Sigma_3 P}$: $A(x) = (\exists w_1)(\forall w_2)(\exists w_3)\ B(x,w_1,w_2,w_3)$
$\mathbf{\Pi_3 P} = \mathbf{co\Sigma_3 P}$: $A(x) = (\forall w_1)(\exists w_2)(\forall w_3)\ B(x,w_1,w_2,w_3)$
$\mathbf{\Sigma_4 P}$: $A(x) = (\exists w_1)(\forall w_2)(\exists w_3)(\forall w_4)\ B(x,w_1,w_2,w_3,w_4)$
$\mathbf{\Pi_4 P} = \mathbf{co\Sigma_4 P}$: $A(x) = (\forall w_1)(\exists w_2)(\forall w_3)(\exists w_4)\ B(x,w_1,w_2,w_3,w_4)$
⋮

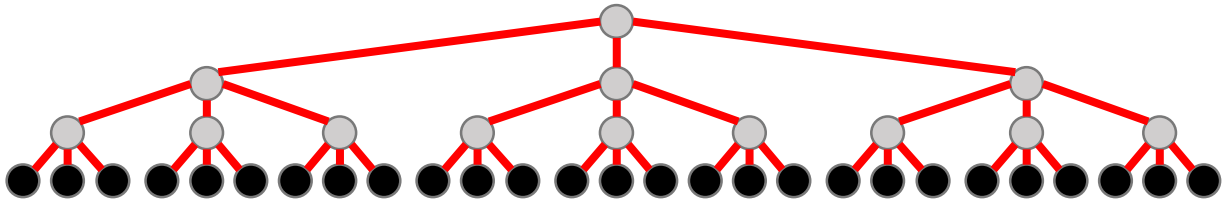
Alternate view of alternating quantifiers

If we have a configuration tree of a nondeterministic program, this helps picture how the leaves need to be colored for various acceptance conditions.

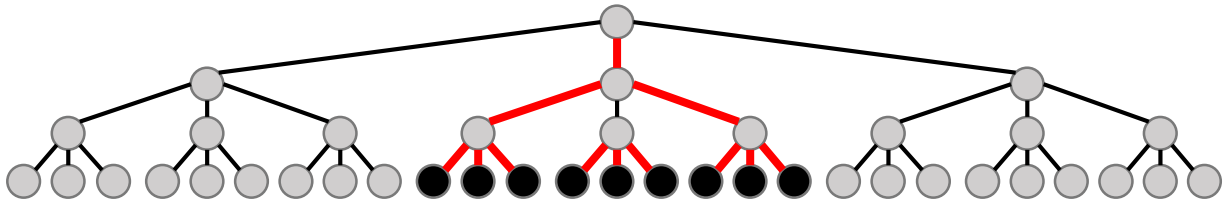
∃ tree



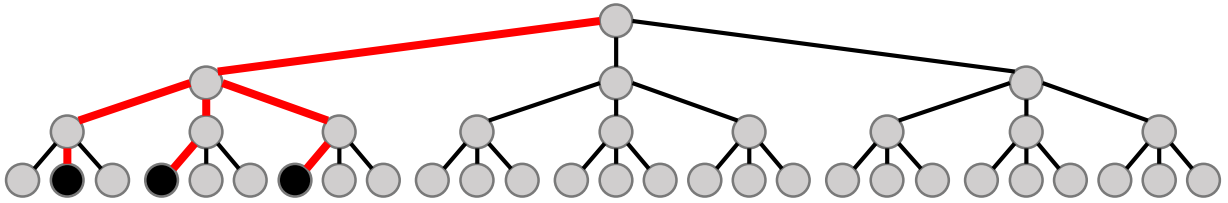
∀ tree



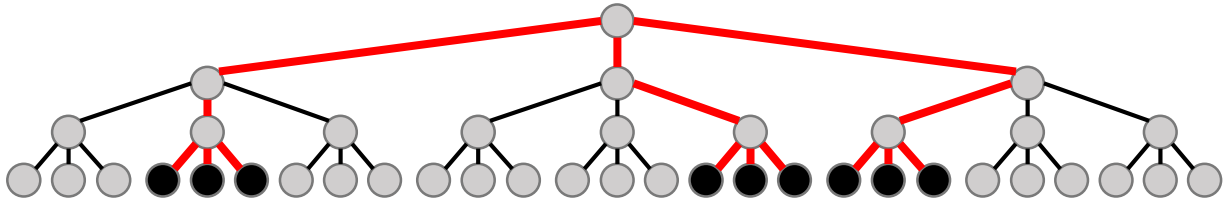
∃∀ tree



∃∀∃ tree



∀∃∀ tree



The polynomial hierarchy

$\boldsymbol{\Sigma_k P}$ / $\boldsymbol{\Pi_k P}$ (a.k.a., "the k^{th} level of the polynomial hierarchy") corresponds (in some sense) to two-player games that last for k moves:

"I have checkmate in k moves" is the same as saying,

k {

"There exists a move I can make, so that
for all moves my opponent could make,
there exists another move I can make, so that
for all moves my opponent could make
…

}

I take my opponent's king"

Simple observations

- Adding a quantifier adds more power but can't reduce it
  - $\mathbf{\Sigma}_k\mathbf{P} \subseteq \mathbf{\Sigma}_{k+1}\mathbf{P}$
  - $\mathbf{\Sigma}_k\mathbf{P} \subseteq \mathbf{\Pi}_{k+1}\mathbf{P}$
  - $\mathbf{\Pi}_k\mathbf{P} \subseteq \mathbf{\Sigma}_{k+1}\mathbf{P}$
  - $\mathbf{\Pi}_k\mathbf{P} \subseteq \mathbf{\Pi}_{k+1}\mathbf{P}$
- ∃ adds a "layer of nondeterminism" asking whether there is a witness that makes the next statement true. In analogy to modifying **P** with "**N**" to get **NP**:
  - $\mathbf{\Sigma}_k\mathbf{P} = \mathbf{N} \cdot \mathbf{\Pi}_{k-1}\mathbf{P}$ … e.g., $C \in$ SMALLER-BOOLEAN-CIRCUIT ⇔ (∃ smaller circuit D) D computes φ_C
  - In particular, defining $\mathbf{\Sigma}_0\mathbf{P} = \mathbf{\Pi}_0\mathbf{P} = \mathbf{P}$, we have $\mathbf{\Sigma}_1\mathbf{P} = \mathbf{NP}$
  - $\mathbf{\Pi}_k\mathbf{P} = \mathbf{co}\mathbf{\Sigma}_k\mathbf{P}$ and $\mathbf{\Sigma}_k\mathbf{P} = \mathbf{co}\mathbf{\Pi}_k\mathbf{P}$ (since negating ∃ gives ∀, and vice versa)

    e.g., !(C is the smallest Boolean circuit computing φ_C)

    ⇔ ![(∀ circuits D)(∃ input w) $|D| < |C| \Rightarrow C(w) \neq D(w)$]

    ⇔ ![(∀ circuits D)(∃ input w) $|D| \geq |C|$ or $C(w) \neq D(w)$]

    ⇔ (∃ circuit D)(∀ inputs w) ![$|D| \geq |C|$ or $C(w) \neq D(w)$]

    ⇔ (∃ circuit D)(∀ inputs w) $|D| < |C|$ and $C(w) = D(w)$

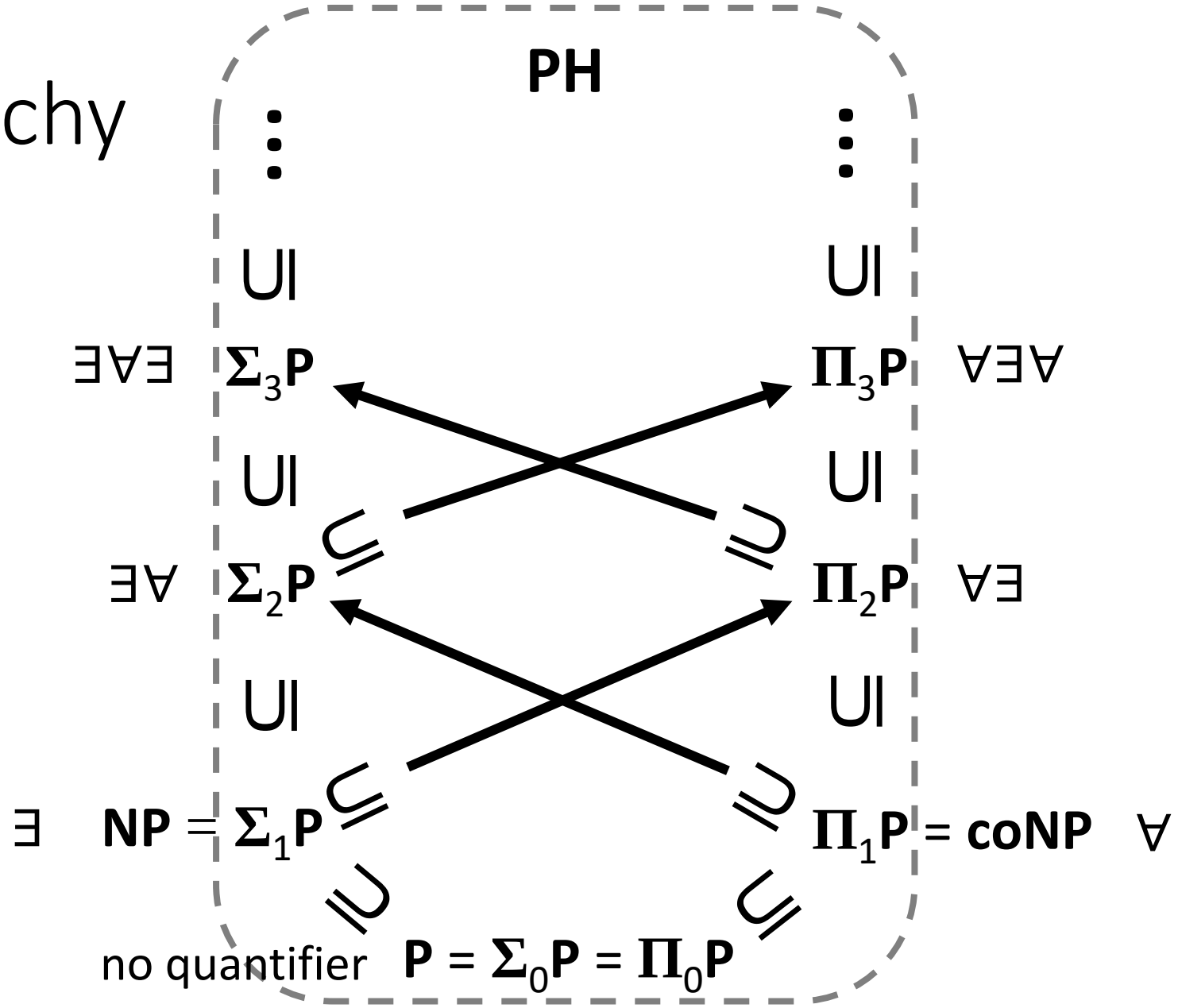
    ⇔ a smaller circuit D computes φ_C

coNP predicate

Polynomial hierarchy

$$\mathbf{PH} = \bigcup_{k=1}^{\infty} \mathbf{\Sigma}_k\mathbf{P}$$

$$= \bigcup_{k=1}^{\infty} \mathbf{\Pi}_k\mathbf{P}$$



Grotesque consequence #1: The Great Collapse

If **P** = **NP**, then **P** = **PH** (i.e., if one ∃ quantifier adds no power, then neither do any additional ∀/∃ quantifiers)

We repeatedly use two facts that hold if **P** = **NP**: (assume $|w|=\text{poly}(|x|)$)

fact (e) For any predicate $B(x,w)$ in **P**, the predicate $B_{\exists}(x) = (\exists w)\ B(x,w)$ is also in **P**.

fact (a) Since **P** = **coP** = **coNP**, the predicate $B_{\forall}(x) = (\forall w)\ B(x,w)$ is also in **P**.

Example: Let $A(x) = (\exists w_1)(\forall w_2)(\exists w_3)\ B_3(x,w_1,w_2,w_3)$, where $B_3 \in \mathbf{P}$, so $A \in \mathbf{\Sigma_3 P}$.

- by **(e)**, the predicate $B_2(x,w_1,w_2) = (\exists w_3)\ B_3(x,w_1,w_2,w_3)$ is in **P**
  $A(x) = (\exists w_1)(\forall w_2)\ B_2(x,w_1,w_2)$
- by **(a)**, the predicate $B_1(x,w_1) = (\forall w_2)\ B_2(x,w_1,w_2)$ is in **P**
  $A(x) = (\exists w_1)\ B_1(x,w_1)$
- by **(e)**, the predicate $B_0(x) = (\exists w_1)\ B_1(x,w_1)$ is in **P**
  $A(x) = B_0(x)$
- so $A \in \mathbf{P}$, showing $\mathbf{P} = \mathbf{\Sigma_3 P}$.

What if **NP = coNP**?

- Exercise : then **NP** = **PH**
- Doesn't necessarily imply that **P** = **NP**, but still seems very unlikely.
- In other words, if **P** ≠ **NP** = **coNP** = **PH**, then although one single ∃ or ∀ quantifier increases computational power, adding more quantifiers would not!
- This is evidence that no problem in **NP** ∩ **coNP** is **NP**-complete (e.g., problems related to factoring), since that would imply **NP** = **coNP** .

Grotesque consequence #2: “Collapse propagates up”

textbook: “*A common misconception is that the **P** vs. **NP** question is about polynomial time. However, it is really a question about the power of nondeterminism in general, and whether finding solutions is harder than checking them, **regardless of how much time we have**. In this section, we will show that if **P** = **NP**, then **TIME**(t(n)) = **NTIME**(t(n)) for essentially any class of functions t(n) that grow at least polynomially.*”

EXP = **TIME**$(2^{\text{poly}(n)})$ = $\bigcup_{c\in\mathbb{N}}$ **TIME**(2^{n^c})

NEXP = **NTIME**$(2^{\text{poly}(n)})$ = $\bigcup_{c\in\mathbb{N}}$ **NTIME**(2^{n^c})

Is **EXP** = **NEXP**?

We don’t know, but if **P**=**NP**, then **EXP** = **NEXP**.

Proof technique: "Padding"

- Assume **P** = **NP**. Let $A \in$ **NEXP**. We'll show that $A \in$ **EXP**.
- Since $A \in$ **NEXP**, there is a (say) 2^{n^3}-time decidable predicate $B(x,w)$ ($n=|x|$) such that

$$A(x)=\texttt{true} \iff \left(\exists w \in \{0,1\}^{2^{n^3}}\right) B(x,w)=\texttt{true}$$

- Consider the following problem C:

  Given: binary $x \in \{0,1\}^n$ and unary $p = 1^{k-n}$

  Question: is $k = 2^{n^3}$? *if so:*

  is there $w \in \{0,1\}^k$ such that $B(x,w)=\texttt{true}$?

  x = 0101000 (n) p = 1111111111111111111111111111111 … 111 (k-n)

  w = 0100100101111010010101001010010100010001011 … 00111 (k)

  Nondeterministic algorithm to solve C :

  - Nondeterministically guess $w \in \{0,1\}^k$ in time k.
  - Run 2^{n^3}-time decider for B on (x,w); also time k.

  Since C's input size is k, C is solvable in nondeterministic linear time, so $C \in$ **NP** = **P**.

- Deterministic algorithm reducing A to C: on input x of length n, compute $C(x, 1^{2^{n^3}-n})$.

  Since $C \in$ **P**, time needed is polynomial in k, e.g., $k^5 = \left(2^{n^3}\right)^5 = 2^{5n^3}$, so exponential in A's input length n. Thus $A \in$ **EXP**.

Time constructible functions

- Generalizing the last proof to other time bounds $t(n)$ means, on an input x of length n, we must write down the pair $(x,1^{t(n)-n})$ in $O(t(n))$ time.
- What if $t(n)$ is itself a computationally difficult function that cannot be computed in time $O(t(n))$?

  e.g., $t(n) = 2n$ if the n^{th} Turing machine halts, $t(n) = 2n+1$ otherwise

  If such pathological time bounds are permitted, then it is possible to prove pathological theorems such as:

  There is a time bound $t(n)$ such that **TIME**$(t(n))$ = **TIME**$\left(2^{2^{2^{t(n)}}}\right)$
- We simply disallow such functions as time bounds.
- Definition: a function $t:\mathbb{N}\rightarrow\mathbb{N}$ is time constructible if there is a program that, on input 1^n, computes $t(n)$ in $O(t(n))$ steps.
- All subsequent time bounds will be assumed time constructible.

Mathematical proofs

> *"Could anything be more unattractive and inelegant? Could any argument be more obfuscatory and unreadable? This isn't mathematics! A proof should be an epiphany from the Gods, not a coded message from the Pentagon. This is what comes from a misplaced sense of logical rigor: ugliness. The spirit of the argument has been buried under a heap of confusing formalism."*

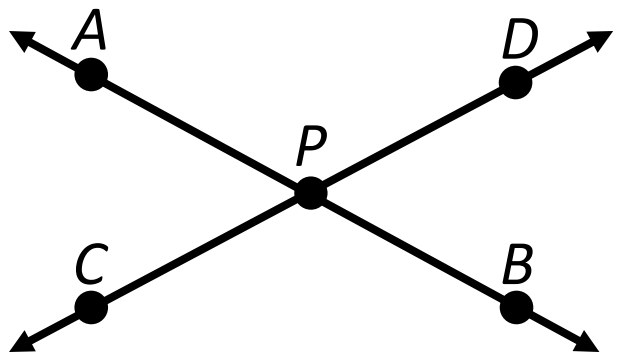
What is a "formal mathematical proof"?

- Not the things we usually write down!
- It is a series of simple steps, each applying an axiom or theorem, e.g.

PROPOSITION 2.1.1. Let $\overline{AB}$ and $\overline{CD}$ intersect at P. Then $\angle APC \cong \angle BPD$.

Proof.

Statement	Reason
1. $m\angle APC + m\angle APD = 180$ $m\angle BPD + m\angle APD = 180$	Angle Addition Postulate
2. $m\angle APC + m\angle APD = m\angle BPD + m\angle APD$	Substitution Property
3. $m\angle APD = m\angle APD$	Reflexive Property of Equality
4. $m\angle APC = m\angle BPD$	Subtraction Property of Equality
5. $\angle APC \cong \angle BPD$	Angle Measurement Postulate



A Mathematician's Lament, Paul Lockhart

Verifying vs. Finding Proofs

Checking proofs is tedious, but simple, so the following problem is in **P**:

PROOF-CHECKING

Given: A mathematical statement *S* and a proof *P*

Question: Is *P* a valid proof of *S*?

So this problem is in **NP**:

SHORT-PROOF

Given: A mathematical statement *S* and an integer *n* in unary

Question: Does *S* have a proof of length *n* or less?

(SHORT-PROOF is **NP**-complete… why? How to show SAT ≤ SHORT-PROOF?)

Grotesque consequence #3: The demise of creativity

- Suppose **P** = **NP**. Write down any unsolved mathematical problem, e.g., Goldbach's Conjecture, Riemann Hypothesis (or any of the Clay problems). Now solve the problem *"Does this statement have a proof with 10^9 steps or less?"*
- Any human-written proof, even tediously expanded to formal detail, will be < 1 billion steps.
- In other words, all of mathematics could be automated. (With machine learning, this may on the horizon anyway.)
- Not just mathematics… most scientific fields repeatedly seek to solve this problem:

  ELEGANT-THEORY:

  Given: a set *E* of experimental data (e.g., astronomical observations) and an integer *n* in unary

  Find: a theory *T* (e.g., Kepler's Laws of planetary motion) of length *n* or less that explains *E*

  textbook: *"No longer do we need a Kepler to perceive elliptical orbits, or a Newton to guess the inverse-square law. Finding these theories becomes just as easy as checking that they fit the data."*

Reasons to Believe **P ≠ NP**

- "Reasons to Believe", Scott Aaronson
  http://www.scottaaronson.com/blog/?p=122
- "The Scientific Case for **P ≠ NP**", Scott Aaronson
  http://www.scottaaronson.com/blog/?p=1720

- A reasonable dissenting view from Dick Lipton
  https://rjlipton.wordpress.com/2011/06/24/polls-and-predictions-and-pnp/
  https://rjlipton.wordpress.com/conventional-wisdom-and-pnp/

Upper Bounds are Easy, Lower Bounds are Hard

- "Upper bound" ≈ "showing a problem has an algorithm of some type" (upper bounding its computational difficulty)
- "Lower bound" ≈ "showing a problem cannot be solved by some class of algorithm" (lower bounding its computational difficulty)
- "Upper bounds are easy"

  To show a problem is in **P**, we need "only" find a single polynomial-time algorithm for it.

- "Lower bounds are hard"

  To show a problem is not in **P**, we need to show that every candidate polynomial-time algorithm fails.

- Now let's look at a few techniques we do have for proving lower bounds

Sorting black boxes

- Mergesort/Heapsort/"median-of-medians"-Quicksort take $O(n \log n)$ time to sort n items in the worst case.
- Can we do better?
- Need to formalize "sorting" a little more:

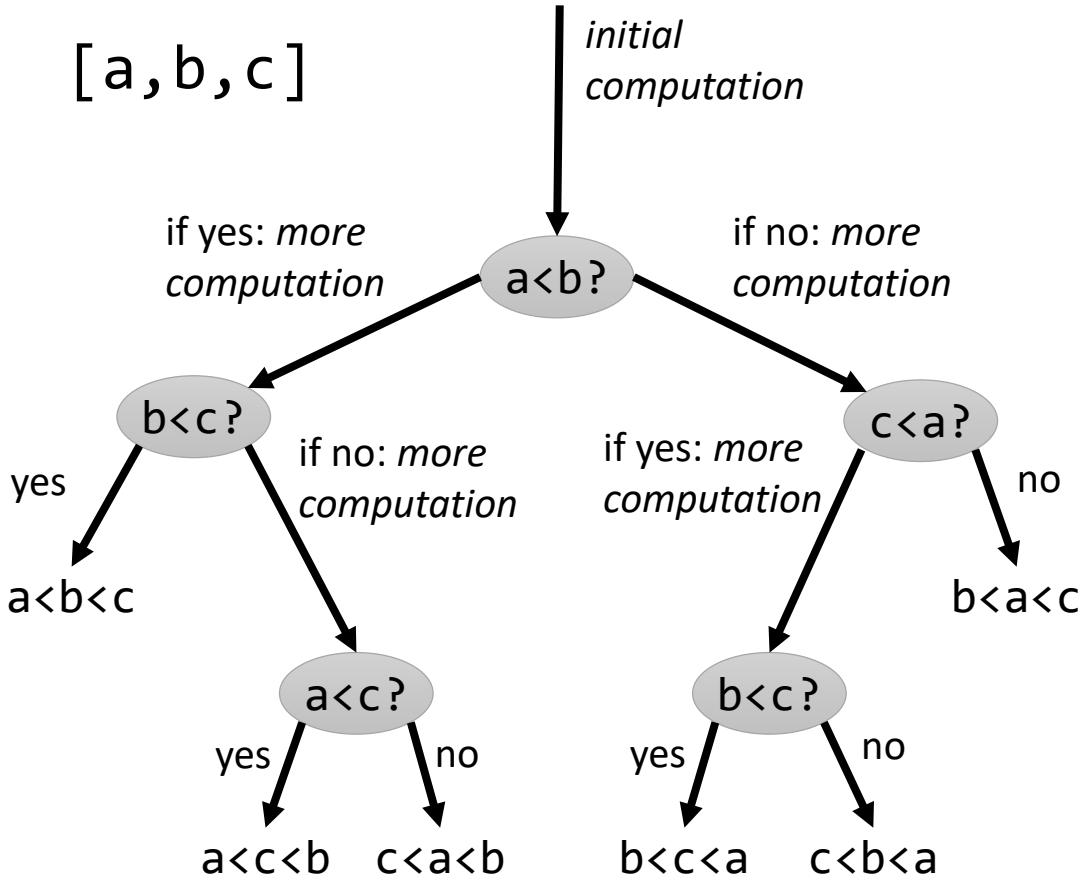
  Task: Sort this input: x = 1010100010011101 10… BUT does it represent a list of:

  - 3-bit integers? 101 010 001 001 110 110 → 001 001 010 101 110 110
  - 2-bit symbols? 10 10 10 00 10 01 11 01 10 → 00 01 01 10 10 10 10 10 11
  - 1-bit bits? 1010100010011101 10 → 0000000001111111 1

- Assume we can't "see the bits" that encode the array. Instead, we can only call a routine `compare(i,j)`, which answers the question "is `array[i] < array[j]`?" (black-box comparison)

Black-box sorting needs $\Omega(n \log n)$ comparisons

Every sorting algorithm, for each n, has an associated decision tree describing the items it will compare in any length-n list:



Each list of length n traverses some branch of this tree.

Worst case # of comparisons = depth of tree

leaves in tree?

$\geq n!$ (# of permutations)

depth of tree $\geq \log n! = \Theta(n \log n)$

Note this shows even if all other computation is instantaneous, # of comparisons $\geq \Theta(n \log n)$

it shows a bound on *information* needed to determine proper sorted order

Are the boxes really black?

- Radix sort can sort n m-bit integers in time $O(nm)$.

  $nm < n \log n$ if there are duplicates.

  Radix sort "looks inside the box" at the bits.

- Black box analogy for 3-SAT: if a 3-CNF formula may only be black-box queried with test assignments, but we can't examine the clauses, then all 2^n assignments must be tested in the worst case.

  3-SAT is not a black box: there is a 1.61^n-time algorithm.

Some problems are provably not in **P**!

- Proving a particular problem is not in **P** is tough… but we can *artificially construct* a problem outside of **P**.

  The trick is to ensure that it is actually in some other complexity class **C**, showing that **C** ≠ **P**.

- We'll construct a problem in **EXP** but not **P**.

  The problem is artificial… this won't tell us whether any particular problem in **EXP** that we care about is actually outside of **P**.

- More generally, we'll show that **TIME**($t(n)$) ⊊ **TIME**($f(n)$) whenever f grows at a rate "sufficiently larger" than t.

- *"With more time, you can compute more."*

  In other words, computational complexity theory is actually worth studying! This whole alphabet soup is not simply a thousand different names for **P**.

Some artificial problems

Let $t(n)$ be a time bound function.

PREDICT$_t$

given: program P and input x

output: $P(x)$ if $P(x)$ halts within $t(|x|)$ steps, otherwise "don't know"

Note:

- t is "hardcoded" into PREDICT$_t$
- $t(|x|)$ steps is an absolute bound, not $O(t(|x|))$

source code of P

CATCH22$_t$

given: program P returning a Boolean

output: $!P(P)$ (negation of $P(P)$) if $P(P)$ halts within $t(|P|)$ steps, otherwise "don't know"

"diagonal" part of the "diagonalization"

inputs

programs

	P_1	P_2	P_3	P_4
P_1	$P_1(P_1)$	$P_1(P_2)$	$P_1(P_3)$	$P_1(P_4)$
P_2	$P_2(P_1)$	$P_2(P_2)$	$P_2(P_3)$	$P_2(P_4)$
P_3	$P_3(P_1)$	$P_3(P_2)$	$P_3(P_3)$	$P_3(P_4)$
P_4	$P_4(P_1)$	$P_4(P_2)$	$P_4(P_3)$	$P_4(P_4)$

A time lower bound

CATCH22$_t$

given: program P returning a Boolean

output: !$P(P)$ (negation of $P(P)$) if $P(P)$ halts within $t(|P|)$ steps, otherwise "don't know"

Then for any program P that halts on input P within $t(|P|)$ steps, CATCH22$_t(P) \neq P(P)$.

Let P_{22} be any program that solves CATCH22$_t$ correctly on all inputs; then in particular CATCH22$_t(P_{22}) = P_{22}(P_{22})$. By the above, P_{22} does not halt on input P_{22} within $t(|P_{22}|)$ steps.

CATCH22$_t$ could still be solvable in (say) $3t(n)$ steps, thus be in **TIME**$(t(n))$. But it is not in **TIME**$(g(n))$ if $g(n) = o(t(n))$.

This also (essentially) shows PREDICT$_t \notin$ **TIME**$(g(n))$, so in general, the fastest way to predict some programs t steps into the future is to actually run them for t steps… no shortcuts

Simulation slowdown

CATCH22$_t$ is designed to be unsolvable by every $t(n)$-time program… but it is solvable. In how much time?

We can simulate $P(P)$ for $t(|P|)$ steps to solve it… doesn't that take only $t(|P|)$ steps and contradict the fact that CATCH22$_t$ cannot be solved in only $t(|P|)$ steps?

No, because the simulation incurs a slowdown… how much? Depends on programming language, so we'll leave the slowdown as a parameter.

Let's say $s(t)$ steps are required to simulate t steps of a program using an interpreter in the same programming language.

- For Turing machines, it's easy to show $s(t) = O(t^2)$.
- A not-obvious proof shows $s(t) = O(t \log t)$ for two-tape Turing machines.
- For programming languages like Python, $s(t) = O(t)$ (but the big-O constant is > 1). For these languages $\mathbf{TIME}(s(t(n))) = \mathbf{TIME}(t(n))$.

Additionally, the simulating program has to set an "alarm clock" that goes off after $t(n)$ steps… so $t(n)$ must be time-constructible.

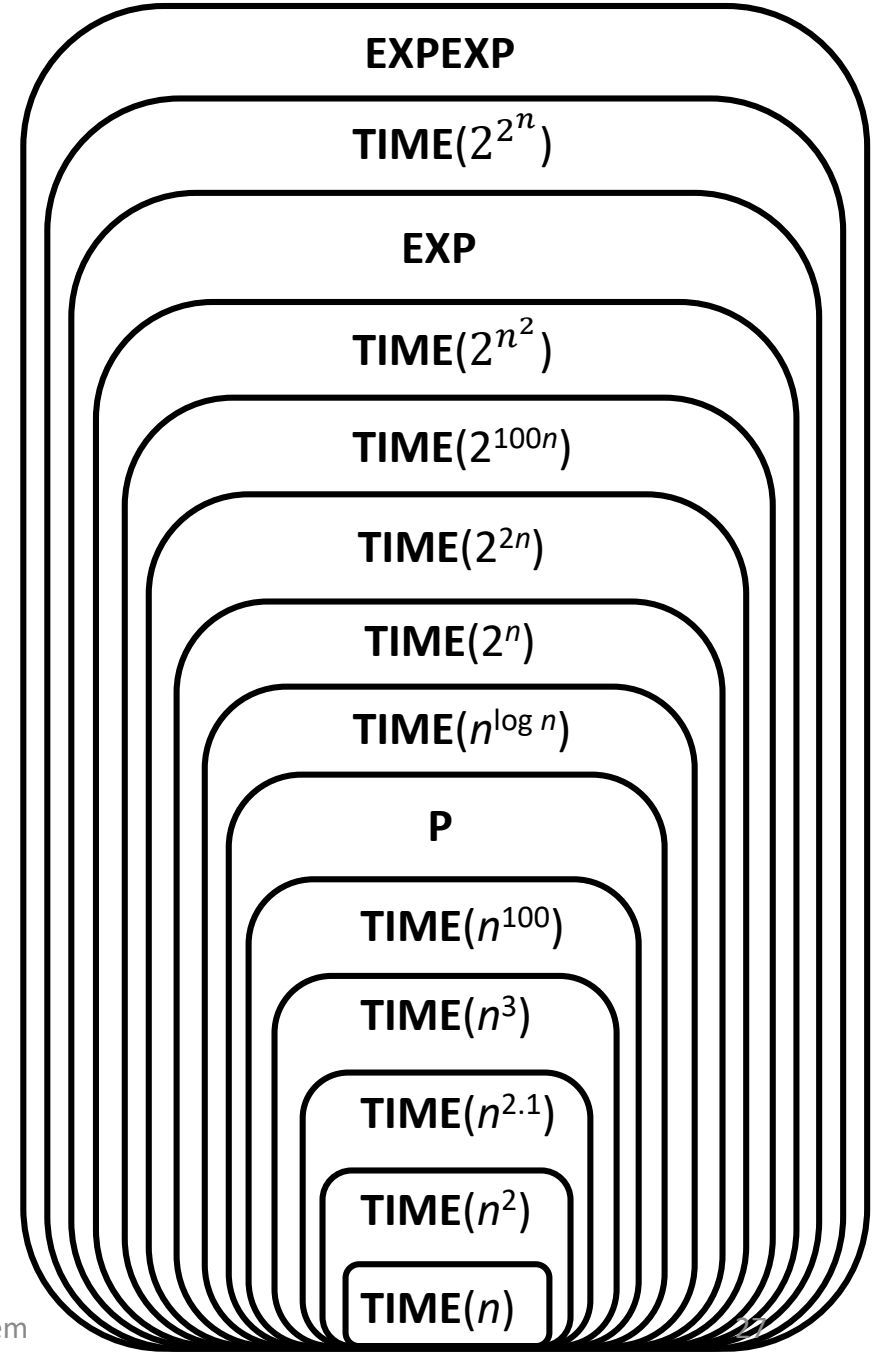
The Time Hierarchy Theorem

Time Hierarchy Theorem: Assume that our programming language can simulate t steps of an arbitrary program, while running an alarm clock that goes off after t steps, in $s(t)$ time. If $t(n)$ is time constructible and $g(n) = o(t(n))$, then $\mathbf{TIME}(g(n)) \subsetneq \mathbf{TIME}(s(t(n)))$.

Proof: $\text{CATCH22}_t \in \mathbf{TIME}(s(t(n)) \setminus \mathbf{TIME}(g(n))$

Corollary for "most" programming languages (where $s(t) = O(t)$… *not* Turing machines):

If $t(n)$ is time constructible and $g(n) = o(t(n))$, then $\mathbf{TIME}(g(n)) \subsetneq \mathbf{TIME}(t(n))$.



P vs. **NP** vs. **EXP**

$$\mathbf{P} \subseteq \mathbf{NP} \subseteq \mathbf{EXP}$$

Since $\mathbf{P} \subsetneq \mathbf{EXP}$, one of the above inclusions is proper, but we don't know which! (probably both are)

<u>Exercise at home</u>:
Let $\mathbf{E} = \bigcup_{c \in \mathbb{N}} \mathbf{TIME}(2^{cn}) \subsetneq \mathbf{EXP}$.
Prove $\mathbf{NP} \neq \mathbf{E}$

Nondeterministic Time Hierarchy Theorem

Theorem: **NTIME**($g(n)$) $\subsetneq$ **NTIME**($t(n)$) whenever $g(n) = o(t(n))$.

Note the lack of "simulation overhead" in this version (even for Turing machines).

So **NP** $\subsetneq$ **NEXP** $\subsetneq$ **NEXPEXP** …

How to prove **P ≠ NP**?

We can separate **P** from **EXP** and **NP** from **NEXP**… why not **P** from **NP**?

Next section shows a "simple" simulation argument provably cannot work.

Complexity theorists' favorite pastime:

Impossibility proofs: proving a certain type of algorithm can't compute something

Complexity theorists' 2nd favorite pastime:

Impossibility of impossibility proofs: proving a certain type of technique can't prove that a certain type of algorithm can't compute something

"For years, people tried unsuccessfully to prove this sort of impossibility result was impossible. Our result shows the impossibility of their goal." –Scott Aaronson

Relativized worlds

- Imagine your programming language has the following magical instruction taking one step to execute:
  - SAT(φ): returns true if and only if φ is a satisfiable Boolean formula
  - We call SAT an **oracle** for this programming language, and the program an **oracle program**.
- This is not like saying "suppose SAT $\in$ **P**":
  - We could imagine SAT $\notin$ **P**, but we can still think about what sort of problems are efficiently solvable by SAT-oracle programs.
  - This is not a contradiction because **P** is defined by *normal* polynomial-time programs, not polynomial-time SAT-oracle programs.
- Alternate way of thinking about it: just implement SAT( ) using the standard slow brute-force search, but then define the **running time relative to SAT** of any algorithm to exclude any time spent in this subroutine.
- $\mathbf{P}^{\text{SAT}}$ = problems solvable in polynomial time relative to SAT; this is a **relativized complexity class**
  - This is **P** in a world in which some website exists that answers SAT queries instantly.

$\mathbf{P}^{\text{SAT}}$

- What problems are in $\mathbf{P}^{\text{SAT}}$?
  - SAT (obviously)
  - $\overline{\text{SAT}}$ (why?)
  - CLIQUE (why?)
  - $\overline{\text{CLIQUE}}$ (why?)
  - any problem in $\mathbf{P}^{\text{CLIQUE}}$ (why?)
- Let $\mathbf{P}^{\mathbf{NP}} = \bigcup_{A \in \mathbf{NP}} \mathbf{P}^{A}$
  - $\mathbf{P}^{\text{SAT}} \subseteq \mathbf{P}^{\mathbf{NP}}$ (obviously)
  - $\mathbf{P}^{\mathbf{NP}} \subseteq \mathbf{P}^{\text{SAT}}$?

$\mathbf{P^{NP}}$ vs. $\mathbf{\Pi_2 P}$

Let ∀∃-SAT be this $\mathbf{\Pi_2 P}$ problem (actually it's $\mathbf{\Pi_2 P}$-complete):

Given: Boolean formula φ with $2n$ inputs $x_1,\ldots,x_n,y_1,\ldots,y_n$

Question: is it the case that $(\forall x_1,\ldots,x_n)(\exists y_1,\ldots,y_n)\ \varphi(x_1,\ldots,x_n,y_1,\ldots,y_n) = 1$?

SAT-oracle program for ∀∃-SAT:

```
Q_A(φ):
  for each x_1,…,x_n in {0,1}^n:
    ψ = n-input circuit obtained by hard-coding x_1,…,x_n into φ
    if !SAT(ψ):
      return false
  return true
```

- Still takes exponential time even with the SAT oracle.
- Our intuition is that this cannot be improved, i.e., $\mathbf{\Pi_2 P}$ has problems not in $\mathbf{P^{NP}}$.
- Homework: $\mathbf{P^{NP}} \subseteq \mathbf{\Sigma_2 P} \cap \mathbf{\Pi_2 P}$

$\mathbf{P}^{\mathbf{NP}}$ vs. $\mathbf{NP}^{\mathbf{NP}}$

TODO: this slide was confusing

- $\mathbf{NP}^{\mathbf{NP}}$-complete problems:
- WITNESS-EXISTENCE^{SAT}:

  Given: a oracle program $P(x,w)$ with the SAT( ) instruction and input x

  Question: is there w with $|w| = \text{poly}(|x|)$ such that $P(x,w)$ accepts?
- CIRCUIT-SAT^{SAT}:

  Given: a Boolean circuit C with AND, OR, NOT, and SAT gates

  Question: is C satisfiable?
- SAT is **NP**-complete, but not $\mathbf{NP}^{\mathbf{NP}}$-complete (probably),
- Homework: $\mathbf{NP}^{\mathbf{NP}} = \mathbf{\Sigma_2 P}$

Relativizing proofs

"Simulation" arguments have a crucial property: they work even if all the programs are oracle programs with access to the same oracle problem A.

- This holds no matter what is the oracle A… could be undecidable, in fact.
- We say such a proof relativizes… all proofs we've seen so far have this property.

Theorems relating complexity classes based on such arguments also show the relativized complexity classes obey the same relationship. So for all oracles A:

- $\mathbf{P}^A \subseteq \mathbf{NP}^A \subseteq \mathbf{EXP}^A$
- $\mathbf{TIME}^A(g(n)) \subseteq \mathbf{TIME}^A(t(n))$ if $g(n) = o(t(n))$
- If an $\mathbf{NP}^A$-complete problem is in $\mathbf{P}^A$, then $\mathbf{P}^A = \mathbf{NP}^A$

Next we'll see a nonmagical justification for thinking about magic oracles:

No proof resolving the **P** vs. **NP** question can relativize.

The point of the magical oracles

Theorem: There are oracles A and B such that $\mathbf{P}^A = \mathbf{NP}^A$ and $\mathbf{P}^B \neq \mathbf{NP}^B$.

Let A = PREDICT:

- Given: program Q, input x, integer t in *binary*
- Question: does $Q(x)$ output "yes" within t steps?

Let $E \in \mathbf{EXP}$, decided by program Q_E in time 2^{n^c}

- We can decide if $x \in E$ with an n^c-time A-oracle program making the query $A(Q_E, x, 2^{|x|^c})$.
- Thus $E \in \mathbf{P}^A$, showing $\mathbf{EXP} \subseteq \mathbf{P}^A$

Let $L \in \mathbf{NP}^A$, decided by a nondeterministic n^c-time A-oracle program Q_L.

- Any query $A(Q, y, t)$ made by Q_L has $t < 2^{n^c}$
- Replace each query with a deterministic subroutine call that executes $Q(y)$ for 2^{n^c} steps.
- This makes Q_L into a nondeterministic $(2^{n^c} \cdot n^c)$-time program (no oracle!), still having only n^c nondeterministic bits.
- To decide L deterministically, iterate over all 2^{n^c} choices for these nondeterministic bits to decide if Q_L accepts. This takes time $2^{n^c} \cdot 2^{n^c} \cdot n^c = O(2^{3n^c})$
- Thus $L \in \mathbf{EXP}$, showing $\mathbf{NP}^A \subseteq \mathbf{EXP}$

$\mathbf{P}^A \subseteq \mathbf{NP}^A$ for all oracles A.

Putting these together, we have $\mathbf{EXP} \subseteq \mathbf{P}^A \subseteq \mathbf{NP}^A \subseteq \mathbf{EXP}$, so they are all equal.

An oracle B where $\mathbf{P}^B \neq \mathbf{NP}^B$

Pick B by the following random experiment:

- for each $n \in \mathbb{N}$, pick a random string $w \in \{0,1\}^n$.
- Set $y \notin B$ for all $y \in \{0,1\}^n \setminus \{w\}$.
- flip an unbiased coin: if Heads, set $w \in B$; if Tails, set $w \notin B$

Define the problem FINICKY-ORACLE^B:

Given: a unary string 1^n

Question: is there $w \in B$ of length n?

FINICKY-ORACLE$^B \in \mathbf{NP}^B$: simply guess $w \in \{0,1\}^n$ in linear time and query $B(w)$.

Let Q be any n^c-time B-oracle program; we show Q (probably) does not solve FINICKY-ORACLE^B.

- On input 1^n, Q queries some list of at most n^c strings; for simplicity assume they are all length n.
- $Q(1^n)$'s output is the same for any oracle B in which all queries return **`false`**.
- Pr[one query returns false] $\geq 1 - \frac{1}{2^n}$, so Pr[all n^c queries return **`false`**] $\geq \left(1 - \frac{1}{2^n}\right)^{n^c} \to 1 \text{ as n} \to \infty$
- Thus with probability almost ½, Q's output is wrong; this happens for some n with probability 1.

With probability 1, FINICKY-ORACLE$^B \notin \mathbf{P}^B$, so $\mathbf{P}^B \neq \mathbf{NP}^B$.

NP-intermediate problems

- Inside **NP**, we've seen a lot of problems that are in **P** or **NP**-complete.
- Are there any others??
- Ladner's Theorem: If **P** ≠ **NP**, then there is a problem in **NP** that is neither in **P** nor **NP**-complete.
- As with the Time Hierarchy Theorem, this is not a *natural* problem; we construct it artificially with the only goals being to ensure that:
  - it's in **NP**
  - it's not in **P**
  - it's not **NP**-complete
- However, it seems likely that *some* natural problems (e.g., factoring) are **NP**-intermediate.

An **NP**-intermediate problem *A*

- SAT(x) = `true` ⇔ x is (a binary encoding of) a satisfiable Boolean formula.
- We will define a function $f:\mathbb{N}\to\mathbb{N}$.
- Define $A(x)$ by $A(x)$ = SAT(x) if $f(|x|)$ is even, `false` if $f(|x|)$ is odd.

f will increase very slowly, so A will look like SAT for long runs, then switch to the constant `false` predicate for long runs, switching forever.

any problem in **P** works here

any **NP**-complete problem works here

By careful design of f, we ensure:

1. f is computable in time polynomial in n, so $A \in$ **NP**,
2. no polynomial-time program solves A, so $A \notin$ **P**, and
3. no polynomial-time program reduces CLIQUE to A, so A is not **NP**-complete.

any **NP**-complete problem works here

x	SAT(x)	$A(x)$	n	$f(n)$
""	true	true	0	0
"0"	false	false	1	0
"1"	false	false		
"00"	true	true	2	0
"01"	false	false		
"10"	true	true		
"11"	false	false		
"000"	true	false	3	1
"001"	true	false		
"010"	true	false		
"011"	false	false		
"100"	false	false		
"101"	true	false		
"110"	false	false		
"111"	true	false		
"0000"	true	true	4	2
"0001"	false	false		
"0010"	true	true		
⋮				

Almost proof

- Assume **P ≠ NP**
- Q_0, Q_1, Q_2, …: enumeration of all polynomial-time programs
- Define A as follows; for each x = "", "0", "1", "00", "01", …

$Q_1(x)=y$ may be before or after x in order

Note dual role of Q_i outputting Boolean (for deciding) and string (for reduction)

Stage 0(a): Let $A(x) = \text{SAT}(x)$.	If $Q_0(x) \neq \text{SAT}(x) = A(x)$,	~~switch to stage 0(b):~~	Q_0 does not solve A
Stage 0(b): Let $A(x) =$ false.	If $\text{CLIQUE}(x) \neq A(Q_0(x))$	~~switch to stage 1(a):~~	Q_0 does not reduce CLIQUE to A
Stage 1(a): Let $A(x) = \text{SAT}(x)$.	If $Q_1(x) \neq \text{SAT}(x) = A(x)$,	switch to stage 1(b):	Q_1 does not solve A
Stage 1(b): Let $A(x) =$ false.	If $\text{CLIQUE}(x) \neq A(Q_1(x))$,	switch to stage 2(a):	Q_1 does not reduce CLIQUE to A
Stage 2(a): Let $A(x) = \text{SAT}(x)$.	If $Q_2(x) \neq \text{SAT}(x) = A(x)$,	switch to stage 2(b):	Q_2 does not solve A
Stage 2(b): Let $A(x) =$ false.	If $\text{CLIQUE}(x) \neq A(Q_2(x))$,	switch to stage 3(a):	Q_2 does not reduce CLIQUE to A

…

Missing details:

- Is A in **NP**?
  - How to enumerate all polynomial-time programs?
- Does each stage end? *need that for all Q_i, for infinitely many x:*
  - $Q_i(x) \neq \text{SAT}(x)$
  - $\text{CLIQUE}(x) \neq A(Q_i(x))$

} both will use hypothesis **P ≠ NP**

Clocked programs

- Let Q_i be the i^{th} program in lexicographical order; we'll enumerate through all of them.
- We only want to enumerate programs that are polynomial time…
  - But we can't simply enumerate all programs and pick out the polynomial time ones… that's undecidable.
  - We only enumerate programs with a "self-alarm-clock": those with a "header" defining constants c and k, that count their steps and quit after cn^k steps.

Example of clocked program

"normal" program

```
def main(a,b):
  d = gcd(a,b)
  print d

def gcd(a,b):
  if b==0:
    return a
  else:
    return gcd(b, a%b)
```

"$3n^7$-clocked" program

```
c = 3
k = 7
time = 0
time_limit = None

def main(a,b):
  n = math.log(a)+math.log(b)
  time_limit = c*(n**k)
  tick()
  d = gcd(a,b)
  tick()
  print d
```

```
def gcd(a,b):
  tick()
  if b==0:
    tick()
    return a
  else:
    tick()
    return gcd(b, a%b)

def tick():
  time += 1
  if time >= time_limit:
    print "time out"
    sys.exit(-1)
```

Clocked programs

- Let Q_i be the i^{th} program in lexicographical order; we'll enumerate through all of them.
- We only want to enumerate programs that are polynomial time…
  - But we can't simply enumerate all programs and pick out the polynomial time ones… that's undecidable.
  - We only enumerate programs with a "self-alarm-clock": those with a "header" defining constants c and k, that count their steps and quit after cn^k steps.
  - This means we miss most polynomial-time programs,
  - BUT… any problem in **P** has infinitely many polynomial-time programs solving it, and *one of them* is a self-clocking program,
  - so even if we include only these, we include all problems in **P**.
- Q_i refers to the i^{th} program matching the above constraint

Two lemmas

Suppose **P** ≠ **NP**. Let Q be a polynomial-time program.

Lemma 1: For infinitely many $x \in \{0,1\}^*$, $Q(x) \neq \text{SAT}(x)$.

why?

Lemma 2: If $A \in \mathbf{P}$, then $(\exists y \in \{0,1\}^*)$ $\text{CLIQUE}(y) \neq A(Q(y))$.

i.e., Q cannot reduce to CLIQUE to A

An **NP**-intermediate problem *A*

Define $A(x) = \begin{cases} \text{SAT}(x) & \text{if } f(|x|) \text{ is even} \\ \texttt{false} & \text{if } f(|x|) \text{ is odd} \end{cases}$

By careful design of *f*, we ensure:

1. *f* is computable in time poly(*n*), so $A \in$ **NP**,
2. $(\forall i)(\exists y)\ Q_i(y) \neq A(y)$
   i.e., no polynomial-time program Q_i solves *A*,
   so *A* is not in **P**
3. $(\forall i)(\exists y)\ \text{CLIQUE}(y) \neq A(Q_i(y))$
   i.e., no polynomial-time program Q_i reduces CLIQUE to *A*,
   so *A* is not **NP**-complete.

n = input length for *A*,
f(*n*) ≈ index of program to "defeat".

As long as *f*(*n*) keeps increasing, conditions (2) and (3) hold.

f(*n*) computed by a self-clocking program:

```
def f(n):
  if n == 0: return 0
  sub = f(n-1)
  if this subroutine hasn't returned after
  n total steps below, return sub
  for all y ∈ {0,1}*:          # in n steps only time
                               # to reach |y| ≤ log n
    i = floor(sub/2)
    if sub is even:                  # stage i(a)
      if Q_i(y) != A(y):
        return sub + 1
    else if sub is odd:              # stage i(b)
      if CLIQUE(y) != A(Q_i(y)):
        return sub + 1
```

$f(n)$ keeps increasing

Suppose **P** ≠ **NP**.

Let Q be a polynomial-time program.

Lemma 1: For infinitely many $y \in \{0,1\}^*$, $Q(y) \neq \text{SAT}(y)$.

Can't get stuck here forever, otherwise Q_i agrees with A on *all strings*, and so agrees with SAT on *all but finitely many strings*, contradicting Lemma 1.

Lemma 2: If $A \in \mathbf{P}$, then $(\exists y \in \{0,1\}^*)$ $\text{CLIQUE}(y) \neq A(Q(y))$.

Can't get stuck here forever, otherwise A would have finitely many yes-instances, thus $A \in \mathbf{P}$, and Lemma 2 says that eventually we'll find a y passing this test.

Define $A(x) = \begin{cases} \text{SAT}(x) & \text{if } f(|x|) \text{ is even} \\ \textbf{false} & \text{if } f(|x|) \text{ is odd} \end{cases}$

```
def f(n):
  if n == 0: return 0
  sub = f(n-1)
  if this subroutine hasn't returned after
  n total steps below, return sub
  for all y ∈ {0,1}*:          # in n steps only time
                               # to reach |y| ≤ log n
    i = floor(sub/2)
    if sub is even:                    # stage i(a)
      if Q_i(y) != A(y):
        return sub + 1
    else if sub is odd:                # stage i(b)
      if CLIQUE(y) != A(Q_i(y)):
        return sub + 1
```

Why is it so hard to prove **P≠NP**? Perhaps…

- Perhaps **P=NP** but the fastest algorithms for SAT take time n^{100}.
  - Maybe the reason all "natural" problems in **P** have running times below n^{10} is that human intuition can only conceive of relatively simple algorithms.
  - I liken this to the unintuitive nature of relativity and quantum mechanics; our intuition was evolved to deal objects of size 1 mm – 1 km, not huge gravity wells or tiny atomic particles, so there's no reason these models *ought* to be intuitive even though they are accurate.
- Perhaps **P=NP** but the proof is nonconstructive: *"There exists an $O(n^c)$-time algorithm for SAT."* (like pigeonhole principle doesn't say *which* hole gets two pigeons)

  So what's the algorithm? *"Proof doesn't say."*

  … Okay then, at least tell me what c is. *"Proof doesn't say."*
- Perhaps the statement **P≠NP** is independent of the axioms of set theory: there is no proof either way.
  - But consider the statement (*): for all $n > 1000$ there is no Boolean circuit with $n^{\log n}$ or fewer gates that solves all CLIQUE instances of size n.
  - If **P=NP** then (*) is false. If **P≠NP** then (*) is "probably" true.
  - If (*) is false then there is a finite (though huge) proof of that fact. So if independent, it's true!