So far

Computational model (Turing machine)

- Class P - efficiently solvable (decision) problems
- Class NP - efficiently verifiable (decision) problems

NP-complete problems: SAT, CoNP

And above... polynomial hierarchy: EXP, NEXP

If $\text{EXP} \neq \text{NEXP}$, then $P \neq NP$

Proof: Padding

Next collection of topics on the agenda

- Is it true that more time makes it possible to solve more problems?
- Are there complexity classes between P and NP?
- Diagonalization
- Oracles
How to prove that two complexity classes are different?

Find a language in one class that is not in the other.

Every language $L \in C$ decided by some TM $M_L$ that runs within resource bound specified by $C$.

Separate $C_1$ and $C_2$ by finding TM $M$ running within resource bounds specified by $C_1$ that differs from every TM in $C_2$ on at least one input.

Then $L = \{ x \mid M(x) = 1 \}$

is a language separating $C_1$ and $C_2$.

$L \in C_1 \setminus C_2$

Essentially only known tool to do this:

DIAGONALIZATION
Recall: Turing machine specified by
- finite alphabet $S$ (symbols)
- finite set of possible states $Q$
- transition function (program) mapping $Q \times S \rightarrow Q \times S \times \{\text{head movement}\}$

Can agree on some encoding of TMs as
(finite) binary strings. Let's use encoding such that
(a) exists "stop marker", and padding with more
bits after stop marker has no effect but encodes
same machine.
(b) "syntax error"-encoding identifies with TM
that immediately halts and rejects, say.

Then

1. Every string $x \in \{0,1\}^*$ represents a TM $M_x$
   Given $i \in \mathbb{N}$ write $M_i$ to denote Turing machine
   encoded by $i$ written in binary.

2. Every TM $M$ is represented by infinitely
   many strings / infinitely many integers.

3. This representation is efficient in that given $x$,
   we can simulate $M_x$ on the universal Turing
   machine with at most a logarithmic overhead.
Write table with rows and columns indexed by integers.

Interpret: Rows $\Leftrightarrow$ TMs
Columns $\Leftrightarrow$ inputs

<table>
<thead>
<tr>
<th>M₁</th>
<th>M₂</th>
<th>M₃</th>
<th>M₄</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

$(i,j)$ contains $M_i$'s output on input $j$ (in binary)

Construct TM by walking diagonally downwards to the left, making sure at least one mismatch per row $\Rightarrow$ Contradiction; such TM can't exist.

**TIME HIERARCHY THEOREM**

If $f, g$ time-constructible functions satisfying $f(n) \log f(n) = o(g(n))$, then

$$\text{DTIME}(f(n)) \subsetneq \text{DTIME}(g(n))$$

Time-constructible?

Technical condition which we won't go into

All "natural" functions $f(n) \geq n$ that you can think of are time-constructible

E.g., $f(n) = n \log n$, $f(n) = n^2$, $f(n) = 2^n$, etc

Will prove:

**TIME HIERARCHY THEOREM, VANILLA VERSION**

$$\text{DTIME}(n) \subsetneq \text{DTIME}(n^{1.5})$$
Proof. Let D be following TM:

On input x, run universal TM U for \(1|x|^{1.4}\) steps to simulate execution of \(M_x\) on x
If U halts with output \(b \in \{0, 1\}\),
output opposite answer \(1 - b\)
Else output 0.

How set time bound for TM? E.g.
- compute \(1|x|\)
- then compute \(1|x|^{1.4}\) & store in counter
- then fill dedicated worktape with special marker symbol
until counter decreased to 0.
- Now move back to start of worktape, start simulation,
and at every step move right on "timer tape"
- abort if ever see non-marker symbol on timer tape

D decides some language, namely \(L_D = \{ x \mid D(x) \uparrow \}\)
D runs in time \(n^{1.4}\) by construction (by factor for simulation
does not change this)
Hence \(L_D \in \text{DTIME}(n^{1.5})\) (by same margin).
We claim \(L_D \notin \text{DTIME}(n)\)

For contradiction, assume \(\exists M\) that on any
x runs in \(\leq c|x|\) steps (for some fixed \(c\))
and outputs \(D(x)\). on any \(x\) \(\leq c'(|x| \log |x|)\) for some \(c'\)

\(M\) can be simulated in time \(O(|x| \log |x|)\) by \(U\). Fix
large enough \(N\) s.t. \(n^{1.4}\) is larger than this if \(n > N\).
Pick some \( x \) of length \( \geq N \) s.t. 
\[ M_x = M \] (possible by (2) above)

Then - on input \( x \), \( D \) will simulate \( M \) on \( x \)
- \( M \) will have time to terminate and output \( M(x) \)
- By def of \( D \) we have \( D(x) = 1 - M(x) + M(x) \)
- But \( M \) decides \( L_D \) by assumption, so \( M(x) = D(x) \)

Contradiction. hence no such \( M \) exists, QED  

There is also a time hierarchy theorem for non-deterministic computation.

**Non-deterministic Time Hierarchy Theorem**

If \( f, g \) are time-construcitble functions satisfying  
\[ f(n+1) = o(g(n)) \], then  
\[ \text{NTIME}(f(n)) \subset \text{NTIME}(g(n)) \]

Proof more subtle. Will skip this.

Most problems studied in \( \text{NP} \) are known either to be in \( \text{P} \) or to be \( \text{NP} \)-complete.

So can it be that every problem in \( \text{NP} \) is either in \( \text{P} \) or \( \text{NP} \)-complete?  
(Results of that flavour known as Dichotomy Theorems.)

Answer: If \( \text{P} = \text{NP} \), then yes (trivially).
If \( \text{P} \neq \text{NP} \), then no, in very strong sense.
What lies between P and NP?

If P = NP, nothing (clearly)

But what if P ≠ NP?

** Ladner’s Theorem **

If P ≠ NP, then there exists a strict, infinite hierarchy of complexity classes between P and NP.

Guided exercise for problem set

- Write vanilla version of this statement
- Most of details can be found in textbook
- Want you to go through the proof and make sure you understand it
- Write nice, complete exposition aimed at student finishing ACK, say
- So practice also writing and presentative skills

** Ladner’s Theorem, Vanilla Version **

If P ≠ NP, then there exists a language \( L \in NP \setminus P \) that is not NP-complete

Caveat: This language \( L \) looks quite convoluted...
But interesting to know it exists.
Main idea: Padding.

Let $P : \mathbb{N} \to \mathbb{N}$ be some function such that $P(n)$ is computable in time polynomial in $n$.

Define $SAT_P$ to be (CNF)SAT with all size-$n$ formulas $\phi$ padded with $n^{P(n)}$ 1's.

$$SAT_P = \{ \phi 0^{n^{P(n)}} | \phi \in \text{CNFSAT} \text{ and } n = |\phi| \}$$

That is: given string $x$, scan from back until first 0. Let $\phi$ be everything before that 0. Set $n = |\phi| = \text{length of } \phi$. Have string $1^k$ after 0.

$x \in SAT_P$ if (a) $\phi \in \text{CNFSAT}$ and (b) $k = n^{P(n)}$

**Observations**

- If $P(n) \in O(1)$, then $SAT_P$ NP-complete
- If $P(n) = \Omega(n/\log n)$, then $SAT_P \in \mathbf{P}$

**Proof:** Problem set

Want to choose padding function in some clever way so that $SAT_P$ is too hard to be in $\mathbf{P}$ (assuming $\mathbf{P} \neq \mathbf{NP}$) but too easy to be NP-complete (because the padding gives extra time).
Here is our padding function $H(n)$

- if $n \leq 4$
  - return 1
- else
  - $i := 0$; failed := TRUE
  - while $i < \log \log n$ and failed
    - failed := FALSE; $i := i + 1$; for all $x \in \{0,1\}^*$ with $|x| \leq \log n$
    - simulate $M_i$ on $x$ for $i \cdot |x|^i$ steps
      - if $M_i$ didn't terminate
        - failed := TRUE
      - else
        - let $b :=$ output of $M_i(x)$
        - split $x = y \cdot 0 \cdot 1^k$ and
        - let $s := 1^{|y|}$
          - recursive call

- check that $b = 1$ if and only if $y \in \text{CNFSAT}$ and $k = s$ $H(s)$
- else
  - failed := TRUE
- endfor
- endwhile
- return $i$
CLAIMS ABOUT $H$

1. $H$ is well-defined (i.e., the algorithm computes a specific function).

2. $H(n)$ is computed in time polynomial in $n$.

3. $SAT_H \in P$ if and only if $H(n) = 0(1)$ [i.e., there exists a $K$ such that $\forall n \ H(n) \leq K$].

4. If $SAT_H \in P$ then $H(n) \to \infty$ as $n \to \infty$.

Proofs: Problem set (plus read Arora-Barak)

Now assume $P \neq NP$.

(i) Suppose $SAT_H \in P$.
Then we can show that CNFSAT $\in P$.
But CNFSAT $NP$-complete. Contradiction.

(ii) Suppose $SAT_H$ $NP$-complete.
Then we can reduce CNFSAT to $SAT_H$ efficiently.
But if so can compose reductions and compress CNFSAT instance so much that they are solvable in polynomial time.
Contradiction.

Detailed proof: Problem set
Are there more interesting and natural non-NP-complete languages in \(\text{NP} \setminus \text{P}\) ?

Obviously, we don't know.

But \textsc{Factoring} and \textsc{GraphIsomorphism} are candidates (though graph isomorphism not so much any longer).