CS154, Lecture 17: 
coNP, Oracles again, Space Complexity
Definition: \( \text{coNP} = \{ L \mid \neg L \in \text{NP} \} \)

What does a coNP computation look like?

In NP algorithms, we can use a “guess” instruction in pseudocode:

\[ \text{Guess string } y \text{ of } |x|^k \text{ length...} \]

and the machine accepts if some \( y \) leads to an accept state

In coNP algorithms, we can use a “try all” instruction:

\[ \text{Try all strings } y \text{ of } |x|^k \text{ length...} \]

and the machine accepts if every \( y \) leads to an accept state
TAUTOLOGY = \{ \phi \mid \phi \text{ is a Boolean formula and every variable assignment satisfies } \phi \}\}

Theorem: TAUTOLOGY is in coNP

How would we write pseudocode for a coNP machine that decides TAUTOLOGY?

How would we write TAUTOLOGY as the complement of some NP language?
Is $P \subseteq \text{coNP}$?

Yes!

$L \in P$ implies that $\overline{L} \in P$ (hence $\overline{L} \in \text{NP}$)

In general, **deterministic** complexity classes are closed under complement
Is \( NP = coNP \)?

It is believed that \( NP \neq coNP \)
Definition: A language $B$ is coNP-complete if

1. $B \in \text{coNP}$

2. For every $A$ in coNP, there is a polynomial-time reduction from $A$ to $B$ ($B$ is coNP-hard)
\[
\text{UNSAT} = \{ \phi \mid \phi \text{ is a Boolean formula and no variable assignment satisfies } \phi \}
\]

**Theorem:** UNSAT is coNP-complete

**Proof:** UNSAT \(\in\) coNP because \(\neg\text{UNSAT} = \text{SAT}\)

(2) UNSAT is coNP-hard:

Let \(A \in\) coNP. We show \(A \leq_p\) UNSAT

On input \(w\), transform \(w\) into a formula \(\phi\) using the Cook-Levin Theorem and an NP machine \(N\) for \(\neg A\)

\[
\begin{align*}
\text{If } w \in \neg A & \Rightarrow \phi \in \text{SAT} & \text{If } w \notin A & \Rightarrow \phi \notin \text{UNSAT} \\
\text{If } w \notin \neg A & \Rightarrow \phi \notin \text{SAT} & \text{If } w \in A & \Rightarrow \phi \in \text{UNSAT}
\end{align*}
\]
\[ \text{UNSAT} = \{ \phi \mid \phi \text{ is a Boolean formula and no variable assignment satisfies } \phi \} \]

Theorem: \text{UNSAT} is coNP-complete

\[ \text{TAUTOLOGY} = \{ \phi \mid \phi \text{ is a Boolean formula and every variable assignment satisfies } \phi \} \]
\[ = \{ \phi \mid \neg \phi \in \text{UNSAT} \} \]

Theorem: \text{TAUTOLOGY} is coNP-complete

1. \text{TAUTOLOGY} \in \text{coNP} (already shown)
2. \text{TAUTOLOGY} is coNP-hard:

\[ \text{UNSAT} \leq_p \text{TAUTOLOGY}: \text{Given formula } \phi, \text{ output } \neg \phi \]
Every NP-complete problem has a coNP-complete counterpart.

NP-complete problems:
SAT, 3SAT, CLIQUE, VC, SUBSET-SUM, ...

coNP-complete problems:
UNSAT, TAUTOLOGY, NOCLIQUE, ...
Is $P = NP \cap \text{coNP}$?
An Interesting Problem in NP ∩ coNP

FACTORIZING
= \{ (m, n) \mid m > n > 1 \text{ are integers, there is a prime factor } p \text{ of } m \text{ where } n \leq p < m \}

If FACTORIZING ∈ P, then we could break most public-key cryptography currently in use!

Theorem: FACTORIZING ∈ NP ∩ coNP
PRIMES = \{n \mid n \text{ is a prime integer}\}

PRIMES is in P
Manindra Agrawal, Neeraj Kayal and Nitin Saxena

Abstract
We present an unconditional deterministic polynomial-time algorithm that determines whether an input number is prime or composite.
Theorem: FACTORING \in NP \cap \text{coNP}

Proof:

The prime factorization \( p_1^{e_1} \ldots p_k^{e_k} \) of \( m \) can be used to efficiently prove that either \((m,n)\) is in FACTORING or \((m,n)\) is not in FACTORING:

First verify each \( p_i \) is prime and \( p_1^{e_1} \ldots p_k^{e_k} = m \)

If there is a \( p_i \geq n \) then \((m,n)\) is in FACTORING
If for all \( i \), \( p_i < n \) then \((m,n)\) is not in FACTORING
NP-complete problems:

SAT, 3SAT, CLIQUE, VC, SUBSET-SUM, ...

coNP-complete problems:

UNSAT, TAUTOLOGY, NOHAMPATH, ...

(NP $\cap$ coNP)-complete problems:

Nobody knows if they exist

P, NP, coNP can be defined in terms of specific machine models, and for every possible machine we can give an encoding of it.

NP $\cap$ coNP is not known to have a corresponding machine model
Polynomial Time With Oracles
Think in terms of Turing Machine pseudocode or a subroutine

An oracle Turing machine $M$ with oracle $B \subseteq \Gamma^*$ lets you include the following kind of branching instructions:

“if ($z$ in $B$) then <do something>
else <do something else>”

where $z$ is some string defined earlier in pseudocode.

By definition, the oracle TM can always check the condition ($z$ in $B$) in one step
Some Complexity Classes With Oracles

\( P^B = \{ L \mid L \text{ can be decided by some polynomial-time TM with an oracle for } B \} \)

\( P^{\text{SAT}} = \text{the class of languages decidable in polynomial time with an oracle for SAT} \)

\( P^{\text{NP}} = \text{the class of languages decidable by some polynomial-time oracle TM with an oracle for some } B \text{ in NP} \)
Is $\mathsf{P^{SAT}} \subseteq \mathsf{P^{NP}}$?
Yes. By definition...

Is $\mathsf{P^{NP}} \subseteq \mathsf{P^{SAT}}$?
Yes:

Every NP language can be reduced to SAT!

For every poly-time TM $M$ with oracle $B \in \mathsf{NP}$, we can simulate every query $z$ to oracle $B$ by reducing $z$ to a formula $\phi$ in poly-time, then asking an oracle for SAT instead.
\[ P_B = \{ L \mid \text{L can be decided by a polynomial-time TM with an oracle for } B \} \]

Suppose \( B \) is in \( P \).

Is \( P_B \subseteq P \)?

Yes

For every poly-time TM \( M \) with oracle \( B \in P \), we can simulate every query \( z \) to oracle \( B \) by simply running a polynomial-time decider for \( B \).

The resulting machine runs in polynomial time
Is $\text{NP} \subseteq \text{P}^{\text{NP}}$?

Yes

*Just ask the oracle for the answer!*

For every $L \in \text{NP}$ define an oracle TM $M^L$ which asks the oracle if the input is in $L$. 
Is \( \text{coNP} \subseteq \text{P}^{\text{NP}} \)?

Yes!

Again, just ask the oracle for the answer!

For every \( L \in \text{coNP} \) we know \( \neg L \in \text{NP} \)

Define an oracle TM \( M^L \) which asks the oracle if the input is in \( \neg L \)

- \text{accept} if the answer is no,
- \text{reject} if the answer is yes

More generally, we have \( \text{P}^{\text{NP}} = \text{P}^{\text{coNP}} \)
$NP^B = \{ L \mid L \text{ can be decided by a polynomial-time nondeterministic TM with an oracle for } B \}$

$coNP^B = \{ L \mid L \text{ can be decided by a poly-time co-nondeterministic TM with an oracle for } B \}$

Is $NP = NP^{NP}$?

Is $coNP^{NP} = NP^{NP}$?

It is believed that the answers are NO
Two Boolean formulas $\phi$ and $\psi$ over the variables $x_1, \ldots, x_n$ are equivalent if they have the same value on every assignment to the variables.

Are $x$ and $x \lor x$ equivalent? Yes

Are $x$ and $x \lor \neg x$ equivalent? No

Are $(x \lor \neg y) \land \neg(\neg x \land y)$ and $x \lor \neg y$ equivalent? Yes

A Boolean formula $\phi$ is minimal if no smaller formula is equivalent to $\phi$.

$\text{MIN-FORMULA} = \{ \phi \mid \phi \text{ is minimal} \}$
Theorem: \textsc{MIN-FORMULA} \in \text{coNP}^{\text{NP}}

Proof:

Define \text{NEQUIV} = \{ (\phi, \psi) | \phi \text{ and } \psi \text{ are not equivalent} \}

Observation: \text{NEQUIV} \in \text{NP} \quad \text{(Why?)}

Here is a \text{coNP}^{\text{NEQUIV}} machine for \textsc{MIN-FORMULA}:

Given a formula \phi,

\text{Try all formulas } \psi \text{ smaller than } \phi:

\text{If } (\phi, \psi) \in \text{NEQUIV} \text{ then } \text{accept} \text{ else } \text{reject}

\textsc{MIN-FORMULA} is not known to be in \text{coNP}
Measuring Space Complexity

We measure space complexity by looking at the largest tape index reached during the computation.
Let $M$ be a deterministic TM.

**Definition:** The space complexity of $M$ is the function $S : \mathbb{N} \rightarrow \mathbb{N}$, where $S(n)$ is the largest tape index reached by $M$ on any input of length $n$.

**Definition:** $\text{SPACE}(S(n)) = \{ L | L \text{ is decided by a Turing machine with } O(S(n)) \text{ space complexity} \}$
Theorem: \textsc{3SAT} \in \text{SPACE}(n)

“Proof”: Try all possible assignments to the (at most $n$) variables in a formula of length $n$. This can be done in $O(n)$ space.

Theorem: \textsc{NTIME}(t(n)) is in \text{SPACE}(t(n))

“Proof”: Try all possible computation paths of $t(n)$ steps for an NTM on length-$n$ input. This can be done in $O(t(n))$ space.
The class $\text{SPACE}(s(n))$ formalizes the class of problems solvable by computers with *bounded memory*.

Fundamental (Unanswered) Question: **How does time relate to space, in computing?**

$\text{SPACE}(n^2)$ problems could potentially take much longer than $n^2$ steps to solve.

*Intuition: You can re-use space, but not time*
Let $M$ be a halting TM that on input $x$, uses $S$ space.

How many time steps can $M(x)$ possibly take?

Is there an upper bound?

The number of time steps is at most the total number of possible configurations!

(If a configuration repeats, the machine is looping.)

A configuration of $M$ specifies a head position, state, and $S$ cells of tape content. The total number of configurations is at most:

$$S |Q| |\Gamma|^S = 2^{O(S)}$$
Corollary:
Space $S(n)$ computations can be decided in $2^{O(S(n))}$ time:

$$\text{SPACE}(s(n)) \subseteq \bigcup_{c \in \mathbb{N}} \text{TIME}(2^{c \cdot s(n)})$$

Idea: After $2^{O(s(n))}$ time steps, a $s(n)$-space bounded computation must have repeated a configuration, so then it will never halt...
PSPACE = \bigcup_{k \in \mathbb{N}} \text{SPACE}(n^k)

EXPTIME = \bigcup_{k \in \mathbb{N}} \text{TIME}(2^{n^k})

PSPACE \subseteq \text{EXPTIME}
Is $P \subseteq \text{PSPACE}$?

YES
Is \( \text{NP} \subseteq \text{PSPACE} \)?

YES
Is $\text{NP}^{\text{NP}} \subseteq \text{PSPACE}$?

YES
\( P \subseteq NP \subseteq PSPACE \subseteq EXPTIME \)

Theorem: \( P \neq EXPTIME \)

Why? The Time Hierarchy Theorem!

\[ \text{TIME}(2^n) \not\subseteq P \]

Therefore \( P \neq EXPTIME \)
Intuition: If you have more space to work with, then you can solve strictly more problems!

Theorem: For functions $s, S : \mathbb{N} \rightarrow \mathbb{N}$ where $s(n)/S(n) \rightarrow 0$

$$\text{SPACE}(s(n)) \subsetneq \text{SPACE}(S(n))$$

Proof IDEA: Diagonalization:
Make a machine $M$ that uses $S(n)$ space and “does the opposite” of all $s(n)$ space machines on at least one input

So $L(M)$ is in $\text{SPACE}(S(n))$ but not $\text{SPACE}(s(n))$