We now embark on a study of computational classes that are more general than NP. As these classes will all contain NP, we do not believe that they represent realistic models of computation. However, they play an important role in the study of several phenomena seemingly unrelated to NP such as randomness, interaction, and counting.

To explain our object of study let's go back to NP and coNP. Back in Lecture 2 we gave some intuition as to why these two complexity classes are believed to be different: While short (polynomial-time verifiable) proofs that, say, a boolean formula is satisfiable obviously exist (in the form of the satisfying assignment), we cannot think of similarly short *refutations* that a formula has no satisfying assignments. So it is reasonable to conjecture that such refutations do not exist, and therefore NP $\neq$ coNP.

1 Beyond NP and coNP

Is there anything interesting beyond NP and coNP? One kind of problem that is hard for both NP and coNP (and therefore unlikely to be in either of them) is a problem that asks for both a proof and a refutation. For example consider the following problem:

> EXACTCNF: Given a boolean formula ϕ in CNF and a number k, is it true that the optimum satisfying assignment of ϕ (the one that satisfies the most clauses) satisfies *exactly* k clauses?

This problem is hard for both NP and coNP. The reason is that it is asks us to both prove something (there exists an assignment that satisfies k clauses) and refute something (no assignment satisfies $k+1$ clauses). However, notice that if P = NP, we can solve this problem in polynomial time: We check if there exist assignments that satisfy at least k and $k+1$ clauses, respectively, and accept if the first answer is "yes" but the second one is "no".

Here is another example. Suppose we are given a circuit C and we are interested if it is minimal – if it is the smallest circuit (say in terms of number of gates) with the given functionality. Formally, C is minimal if:

> MINCKT: For every circuit C' that is smaller than C, there exists an input x such that $C'(x) \neq C(x)$.

This is a universally quantified statement, so it seems to be of the coNP type: To refute this statement, we need to provide a circuit C' that is smaller than C but has the same functionality. However, we do not know how to efficiently verify that C and C' have the same functionality. It seems that to do so, we need to go over all assignments of C and C', which takes exponential time.

The problem MINCKT is also hard for both NP and coNP, but it seems to involve even an extra level of hardness not present in EXACTCNF: Here the universal and existential quantifiers are nested, so the input x depends on the circuit C'. So even if we assumed that P = NP, it is not clear this would help us solve MINCKT.

It turns out that this can be done anyway. To show how let's have some definitions.

2 The polynomial-time hierarchy

These problems MINCKT and EXACTCNF are examples of problems in the *polynomial-time hierarchy*. We start describing come classes in the hierarchy:

Definition 1. The class Σ_2 consists of those decision problems L for which there exists a polynomial-time TM A and a polynomial p such that $x \in L$ if and only if there exists y of length $p(|x|)$ such that for all z of length $p(|x|)$, $A(x, y, z)$ accepts.

The class Π_2 consists of those decision problems L for which there exists a polynomial-time TM A and a polynomial p such that $x \in L$ if and only if for all y of length $p(|x|)$ there exists a z of length $p(|x|)$ such that $A(x, y, z)$ accepts.

It is obvious that MINCKT is in Π_2. What about EXACTCNF? We can restate this problem as follows:

> (There exists x that satisfies k clauses of ϕ) and (for all y, y does not satisfy $k + 1$ clauses of ϕ).

Since the two quantifiers are independent, they can be nested in either way, so EXACTCNF is in both Σ_2 and Π_2.

We can generalize the definitions of Σ_2 and Π_2 to obtain complexity classes Σ_k and Π_k for larger k. The class Σ_k consists of those decision problems L for which there exists a polynomial-time TM A and a polynomial p such that

$$x \in L \Leftrightarrow \exists y_1 \forall y_2 \cdots \exists / \forall y_k \; A(x, y_1, y_2, \cdots, y_k) \text{ accepts, where } |y_i| = p(|x|), i = 1, \cdots k$$

and Π_k is defined in a similar way. The *polynomial-time hierarchy* consists of the union of all of these classes. Most of our intuition about the polynomial-time hierarchy comes from NP (i.e. Σ_1), coNP (i.e. Π_1), Σ_2 and Π_2, so we'll focus on those.

By definition, Σ_k consists of those problems whose complements are in Π_k, and we have the following containments:

$$\mathrm{P} \subseteq \mathrm{NP}, \mathrm{coNP} \subseteq \Sigma_2, \Pi_2 \subseteq \Sigma_3, \Pi_3 \subseteq \cdots \subseteq \mathrm{EXP}.$$

We believe that all of these containments are distinct, although we only know for sure that $\mathrm{P} \neq \mathrm{EXP}$.

3 Relations within the polynomial-time hierarchy

We are now in a position to show that if P = NP, then the whole polynomial-time hierarchy collapses to P. We will just show this is true for Σ_2. The generalization to higher levels is straightforward.

Theorem 2. *If* P = NP*, then* Σ_2 = P

Proof. Take any L in Σ_2. Then

$$x \in L \Leftrightarrow \exists y \forall z, A(x, y, z) \text{ accepts, } |y| = |z| = p(|x|)$$

for some polynomial-time TM A and polynomial p. Now define the language L' by

$$(x, y) \in L' \text{ if } \forall z,\ V(x, y, z) \text{ accepts, } |z| = p(|x|).$$

Since P = NP, coNP = P, so L' is in P. So there is a polynomial-time TM M for L', and

$$x \in L \Leftrightarrow \exists y, M(x, y) \text{ accepts, } |y| = p(|x|)$$

and L is in NP. Since NP = P we get $L \in$ P. □

Using a very similar argument we can also show that if NP = coNP, then Σ_2 = NP, and the whole hierarchy collapses to NP. All these results extend to higher levels of the hierarchy. We summarize them in this table:

	P, NP, coNP	PH
Beliefs	NP ≠ coNP	$\Sigma_k \neq \Pi_k$ for all k
Facts	P = NP ⇒ coNP = P	P = NP ⇒ $\Sigma_k = \Pi_k$ = P, for all k $\Sigma_k = \Pi_k \Rightarrow \Sigma_{k'} = \Pi_{k'} = \Sigma_k$, for $k' \geq k$

4 Oracles

An *oracle* in complexity theory is an imaginary subroutine. For example, an oracle for SAT is an imaginary procedure that on input a boolean formula ϕ, outputs 1 if ϕ is satisfiable, and 0 otherwise. We do not believe that such a procedure can be implemented efficiently; however imagining that we have it allows us to study the power we could get from an efficient SAT solver, if it existed. In general, an oracle can be an arbitrary function $O : \{0,1\}^* \to \{0,1\}^*$, and we are interested in what we could possibly achieve if we had efficient access to the functionality of O.

An *oracle Turing Machine* is a Turing Machine which contains two special tapes, a *query tape* and a (read-only) *answer tape* and has a special *query state*. The computation of this TM proceeds as usual, except when the TM goes into the query state. Then the contents of the answer tape are updated solely as a function of the query tape. This step takes one unit of computation time. The TM then goes to some other state and the computation proceeds as usual.

The relationships between queries (specified on the query tape) and answers (provided on the answer tape) is specified by the oracle O. When an oracle TM is instantiated with oracle O, if the

query tape contains the string $q \in \{0,1\}^*$, going to the query state will result in the answer tape being overwritten by the string $a = O(q)$. We use the notation $M^?$ to describe an oracle TM, and M^O for oracle TM $M^?$ instantiated by the oracle O.

To think of oracle Turing Machines, consider the following scenario. You are writing some big computer program, but you depend on the work on your partner, who has to write an important subroutine. The functionality of your program is then not completely determined by your work, but also by what your partner will provide. For different functionalities O provided by your partner, your program M^O will behave in a different way. This is what is captured by the notation M^O.

We already saw a very special kind of oracle TM: reductions. For example when we showed that CSAT reduces to SAT we showed how to transform an instance C of CSAT into one ϕ of SAT so that an oracle that determines if $\phi \in$ SAT tells us immediately if $C \in$ CSAT. So the reduction from CSAT to SAT can be viewed as a polynomial-time oracle TM that, when given oracle access to SAT, solves CSAT. However, not every oracle TM can be described as a reduction: A reduction is special that it makes exactly one call to the oracle and outputs as its answer whatever was provided by the oracle.

4.1 Oracle complexity classes

The main reason we are interested in oracles is that they will allow us to study how computations behave when they are given some extraordinary power, for instance the ability to solve SAT. To understand this power we need to talk not about single oracle TMs, but about all possible efficient oracle TMs. This motivates the following definition.

Definition 3. Let O an oracle. The class P^O consists of all decision problems that are decided by some polynomial-time oracle TM $M^?$ when $M^?$ is instantiated with the oracle O.

Similarly, we can define the class NP^O: This is the class of decision problems that are decided by some nondeterministic polynomial-time oracle TM $M^?$ when $M^?$ is instantiated with the oracle O. Now for the same reason that P is contained in NP, we get that P^O is contained in NP^O.

We can do this for other complexity classes too: Any time we have a complexity class $\mathbf{C}$ which is the class of problems solved by some class of Turing Machines and O is an oracle, we can define the analogous class $\mathbf{C}^O$ which is solved by oracle Turing Machines of the same type when instantiated with the oracle O.

Let's play with oracles for a bit to get a feel about what they do. For instance, what is $\mathrm{P}^{\mathrm{MATCHING}}$? This is just P, since any call to the oracle can be simulated by the polynomial-time machine making the call. What about $\mathrm{NP}^{\mathrm{MATCHING}}$? Again, any call to the oracle can be simulated by the machine, so $\mathrm{NP}^{\mathrm{MATCHING}} = \mathrm{NP}$.

Let's now try some more powerful oracles that solve harder problems. What about $\mathrm{P}^{\mathrm{SAT}}$? A polynomial-time machine with a SAT oracle can solve any NP question: first reduce to SAT, then ask the question. But it can also solve any coNP question: reduce to SAT, ask the question, then output the opposite answer. Therefore $\mathrm{P}^{\mathrm{SAT}}$ contains both $\supseteq$ NP and coNP. The problem EXACTSAT is an example of a problem in $\mathrm{P}^{\mathrm{SAT}}$.

4.2 Oracles and the polynomial-time hierarchy

How does $\mathrm{P}^{\mathrm{SAT}}$ relate to Σ_2 and Π_2? The answer is given by the following theorem:

Theorem 4. $\mathrm{NP}^{\mathrm{SAT}} = \Sigma_2$

Since $\mathrm{P}^{\mathrm{SAT}} \subseteq \mathrm{NP}^{\mathrm{SAT}}$, we get that $\mathrm{P}^{\mathrm{SAT}} \subseteq \Sigma_2$. But $\mathrm{P}^{\mathrm{SAT}}$ is closed under complement, so we also get $\mathrm{P}^{\mathrm{SAT}} \subseteq \Pi_2$.

Proof. We first show that $\Sigma_2 \subseteq \mathrm{NP}^{\mathrm{SAT}}$. Let L be a Σ_2 problem and let A be a polynomial-time TM such that

$$x \in L \Leftrightarrow \exists y \forall z \ A(x, y, z) \text{ accepts, } |y| = |z| = p(|x|)$$

for some polynomial p. To show that $L \in \mathrm{NP}^{\mathrm{SAT}}$, we will simulate the computation of A by an oracle NP machine $N^?$ (which expects a SAT oracle). The nondeterministic tape of N will correspond to the string y. So, given x and y, $N^?$ needs to check if for all z, $A(x, y, z)$ accepts. To do so, N wants to ask its oracle the question "Does $A(x, y, z)$ reject for some z?" and output the opposite answer. Since this is an NP-type question, it can be posed to a SAT oracle.

More formally, $N^?$ does the following: On input x and nondeterminism y, it constructs a TM $Q_{x,y}$ that on input z accepts iff $A(x, y, z)$ rejects. It converts $Q_{x,y}$ into a CNF formula ϕ via the Cook-Levin reduction (which can be implemented in polynomial time) and asks its oracle the query ϕ. $N^?$ then outputs the opposite answer from the one received by the oracle (since the oracle tells it if for some z, $A(x, y, z)$ rejects, and $N^?$ wants to tell if for all z, it accepts.)

We now argue that $\mathrm{NP}^{\mathrm{SAT}} \subseteq \Sigma_2$. Now let $L \in \mathrm{NP}^{\mathrm{SAT}}$ and let $N^?$ be an oracle TM that, when given access to a SAT oracle, solves L. We will now construct the NTM A. Let's think of the three parts of the input to A as being divided into an x-tape, a y-tape, and a z-tape. The y-tape of A will contain the nondeterminism of N. As $N^?$ starts its computation, A will follow along. The question is what happens when $N^?$ makes an oracle call.

Since A cannot make oracle calls, it will do the following. When $N^?$ makes its ith oracle call with the query ϕ_i, A will remember ϕ_i and will guess the answer b_i to ϕ_i on its y-tape. It will then proceed pretending that the answer was correct. At the end of the computation, it will need to check that $\mathrm{SAT}(\phi_i) = b_i$ for all the queries asked. For those $b_i = 1$, this involves asking if there exists some assignment a_i such that $\phi_i(a_i) = 1$. For those $b_i = 0$, it involves asking if for all assignments a_i', $\phi_i(a_i') = 0$. So the guesses for a_i can be included on the y-tape, while those for a_i' can be included on the z-tape of A.

Reformulating, A checks the following: Does there exist a string y, bits $b_1, \ldots, b_n$, and strings $a_1, \ldots, a_n$ such that for all strings $a_1', \ldots, a_n'$ it holds that, $N^?$ accepts on nondeterminism y, queries ϕ_i and answers b_i and either $b_i = 1$ and $\phi_i(a_i) = 1$ or $b_i = 0$ and $\phi_i(a_i') = 0$? □

This argument gives an alternate characterization of the polynomial hierarchy using oracle machines, and it can be extended to higher levels of the hierarchy too.

5 Circuits and the polynomial-time hierarchy

When we talked about boolean circuits, we said that while circuits are more powerful than Turing Machines, we do not think this power is at all useful for solving NP problems. Indeed, proving that NP $\not\subseteq$ P and the more general NP $\not\subseteq$ P/poly are regarded as having a similar level of difficulty. One way to formalize this would be to say that if NP $\subseteq$ P/poly, then NP = P. While we cannot say this, we can prove a related statement regarding the polynomial-time hierarchy:

Theorem 5. *If* NP $\subseteq$ P/poly*, then* $\Sigma_2 = \Pi_2$.

To prove this theorem, it will be convenient to use the following Π_2-complete problem. Its completeness for Π_2 follows directly from the fact that SAT is complete for NP:

$$\forall_2\text{SAT} = \{\phi\colon \phi \text{ is a CNF s.t. } \forall y \exists z\colon \phi(y,z)\}.$$

Proof. We will assume that NP $\subseteq$ P/poly and prove that $\forall_2$SAT is in Σ_2.

The assumption NP $\subseteq$ P/poly says that there is a polynomial-size circuit family $\{C_n\}$ that decides SAT. What we will need is a polynomial-size circuit family for the *search* version of SAT: This is a family of circuits $\{S_n\}$ with multiple bits of output so that on input a CNF ϕ of length n, $S_n(\phi)$ outputs some satisfying assignment for ϕ, if one exists (and is allowed to output anything otherwise).

We can deduce the existence of the polynomial-size circuit family $\{S_n\}$ by a search-to-decision reduction similar to the one from Homework 1: On input ϕ of length n, S_n produces CNFs ϕ_0 and ϕ_1 by setting the first input bit x_1 to 0 and 1, respectively. It then uses the appropriate circuit from $\{C_n\}$ to determine if ϕ_0 or ϕ_1 are satisfiable. If either one of them is satisfiable, say ϕ_b, then it outputs b as the first bit of a satisfying assignment and continues the same process to recover the rest of the assignment. If neither is satisfiable, S_n can output an arbitrary output.

Now let $\phi_a(z)$ be the CNF obtained by fixing y to the value a in $\phi(y,z)$. If $\phi_a(z)$ (assume its length is n) is satisfiable, then $S_n(\phi_a)$ will output a satisfying assignment for it. In other words, for any $y = a$ we have that

$$\exists z\colon \phi(a,z) \qquad \text{if and only if} \qquad \phi(a, S_n(\phi_a)) \text{ is true.}$$

Since the value of a is arbitrary, it follows that

$$\forall a \exists z\colon \phi(a,z) \qquad \text{if and only if} \qquad \forall a\colon \phi(a, S_n(\phi_a)) \text{ is true.}$$

And therefore

$$\phi \in \forall_2\text{SAT} \iff \forall a\colon \phi(a, S_n(\phi_a)).$$

We are making progress: We got rid of an existential quantifier! In fact the last statement looks like a coNP statement. However it is not: It is unclear how the condition $\phi(a, S_n(\phi_a))$ can be checked in polynomial time because S_n is a circuit.

While we do not know the circuit S_n, we know it must exist and have size $p(n)$ for some polynomial p. So we can try to guess it nondeterministically: Let's try all circuits of size $p(n)$ and see which one of them works:

$$\phi \in \forall_2\text{SAT} \iff \exists C \text{ of size } p(n)\ \forall a\colon \phi(a, C(\phi_a))$$

If the formula ϕ_a is satisfiable for all a, then the circuit S_n will be guessed on some computation path so $S_n(\phi_a)$ will produce a satisfying assignment z for $\phi(a, z)$. If ϕ_a is not satisfiable for some a, then $\phi(a, z)$ is false for all z, so no matter which C we choose, $C(\phi_a)$ will be false. □

6 Randomness and the polynomial-time hierarchy

Theorem 5 in particular implies that if NP $\subseteq$ BPP, then $\Sigma_2 = \Pi_2$, giving more evidence to our belief that randomness is not too powerful: Unless the polynomial-time hierarchy collapses, NP-complete problems do not have efficient randomized algorithms. But are there problems outside NP that can be solved by randomized algorithms?

In Lecture 6 we saw that assuming certain circuit lower bounds, we can in fact show that BPP = P, so in particular BPP is contained in NP. However, BPP $\subseteq$ NP appears to be a weaker statement than BPP = P. Can we prove it without making any assumptions? This is not known; but once we go one level up the polynomial-time hierarchy things get easier:

Theorem 6. BPP $\subseteq \Sigma_2 \cap \Pi_2$.

It is in fact sufficient to prove that BPP $\subseteq \Sigma_2$, since BPP is closed under complementation.

To explain the proof of this theorem, let's recall the Nisan-Wigderson pseudorandom generator. This is a device that we can use to simulate the randomness of any randomized algorithm M that runs in time $p(n)$ for any polynomial $p(n)$. We assumed that we have a "hard" family of functions $\{f_m\colon \{0,1\}^m \to \{0,1\}\}$ where f_m can be computed in time $2^{O(m)}$, but no circuit of size $2^{\delta m}$ can compute f_m correctly even on a $1/2 + 2^{-\delta m}$ fraction of inputs, where $\delta > 0$ is some constant. The randomized simulation operates as follows: On input x of length n, we extract $p(n)$ pseudorandom bits by running the Turing Machine that computes f_m on some carefully chosen collection of substrings from some "input seed" s of size $O(\log n)$. Notice that $m = O(\log n)$. Then we proved that from the point of view of M, for any fixed input x, the bits we extracted in this way behave almost like random bits (with error strictly less than $1/6$), so by enumerating over all $2^{O(\log n)}$ choices for s, we can determine whether $M(x)$ accepts or rejects for more than $2/3$ values of the randomness.

Now suppose we dropped the assumption that $\{f_m\colon \{0,1\}^m \to \{0,1\}\}$ is computable in time $2^{O(m)}$. Then it turns out that we can get the second assumption for free:

Lemma 7. *There is a constant $\delta > 0$ such that for sufficiently large m, there exists a function $f : \{0,1\}^m \to \{0,1\}$ such that every circuit C on m inputs of size $2^{\delta m}$, $\Pr_{x\sim\{0,1\}^m}[C(x) = f(x)] \leq 1/2 + 2^{-\delta m}$.*

Proof. Let f be a random function from $\{0,1\}^m \to \{0,1\}$. We show that $\Pr_{x\sim\{0,1\}^m}[C(x) = f(x)] \leq 1/2 + 2^{-\delta m}$ for every C with nonzero probability over the choice of f. Let's first look at a particular

C. Since f is random, the events $C(x) = f(x)$ have probability $1/2$ each and are independent as x ranges over $\{0,1\}^m$. The expected number of x such that $C(x) = f(x)$ is exactly $2^m/2$. Using the Chernoff bound, the probability that $C(x) = f(x)$ for more than $(1+\varepsilon)2^m/2$ values of x is at most $2^{-\varepsilon^2 \cdot 2^m/3}$. Setting $\varepsilon = 2 \cdot 2^{-\delta m}$ we get that

$$\Pr_f\left[\Pr_{x\sim\{0,1\}^m}[C(x) = f(x)] > 1/2 + 2^{-\delta m}\right] \leq 2^{-2^{(1-2\delta)m}}.$$

Recall that the number of circuits of size s is at most 2^{s^2}. For $s = 2^{\delta m}$ we get that

$$\Pr_f\left[\Pr_{x\sim\{0,1\}^m}[C(x) = f(x)] > 1/2 + 2^{-\delta m} \text{ for some } C\right] \leq 2^{-2^{(1-2\delta)m}} \cdot 2^{2^{2\delta m}} < 1$$

as long as $\delta < 1/4$. □

Can we simulate the randomness of M using this assumption only? Here is one idea: On input x of length n, set $m = c \log n$ where c is a sufficiently large constant (depending on the running time $p(n)$ of M) and try to find *some* hard function $f\colon \{0,1\}^m \to \{0,1\}$. We can describe this function as a table of values $f(x)$ for all $x \in \{0,1\}^m$. If we manage to get hold of such a function, then we can turn this function into a pseudorandom generator via the Nisan-Wigderson construction and use it to simulate the randomness of $M(x)$.

But how much time does it take to find the "hard" function f_m? This may involve potentially searching over all 2^{2^m} functions from $\{0,1\}^m$ to $\{0,1\}$. Although $m = O(\log n)$, this process may still take time exponential in n.

However, remember that we are not aiming to show that BPP = P, but the weaker statement BPP $\subseteq \Sigma_2$. And the function f_m can be described using a "Σ_2-sentence":

> There exists $f : \{0,1\}^m \to \{0,1\}$ such that for every C of size at most $2^{\delta m}$, the number of inputs z such that $C(z) = f(z)$ is at most $2^{m-1} \cdot (1/2 + 2^{-\delta m})$.

Since $m = O(\log n)$, the size of both f and C are at most poly(n), so this sentence looks exactly like a Σ_2 type computation. To finish up the proof, the Σ_2-simulation of M can be described like this:

> On input x of length n, set $m = c \log n$ and accept iff there exists $f : \{0,1\}^m \to \{0,1\}$ such that for every C of size at most $2^{\delta m}$, the number of z such that $C(z) = f(z)$ is at most $2^{m-1} \cdot (1/2 + 2^{-\delta m})$, and $M(x, G(z))$ accepts for more than half of the strings $G(z), z \in \{0,1\}^{O(m)}$, where G is the Nisan-Wigderson pseudorandom generator construction based on f.

There is also a different proof of this theorem that does not make use of the Nisan-Wigderson generator.