Let's try another reduction

$$SAT \leq_p 3\text{-}SAT$$

- Very common type of reduction from a general to specific version of a problem.
- Recall, the difference between SAT and 3-SAT is that in 3-SAT each clause has 3 literals.

Reduction

Given an instance I of SAT, take every clause with ≤ 3 literals as is (note, you can always "pad" clauses with < 3 literals to have 3). Let $C = (a_1 \vee a_2 \vee \ldots \vee a_k)$ be a clause with $k > 3$ literals. Create a new set of clauses in the 3-SAT instance:

$$(a_1 \vee a_2 \vee y_1) \wedge (\bar{y}_1 \vee a_3 \vee y_2) \wedge (\bar{y}_2 \vee a_4 \vee y_3) \ldots (\bar{y}_{k-3} \vee a_{k-1} \vee a_k)$$

Where each of the y_i are new variables. Call the resulting 3-SAT instance I'. The conversion of I to I' is clearly polynomial-time.

Claim: There is a satisfying assignment for $C = (a_1 \vee a_2 \vee \dots \vee a_k)$ iff is a satisfying assignment for $C' = (a_1 \vee a_2 \vee y_1) \vee (\bar{y}_1 \vee a_3 \vee y_2) \dots$

Proof: C' has a satisfying assignment $\rightarrow$ C has a satisfying assignment

$\Rightarrow$ If C' is satisfied, then at least 1 of $a_1, \dots, a_k$ is true; otherwise, y_1 would have to be true, which would force y_2 to be true ... so that eventually $\bar{y}_{k-3}$ would be false, along with the whole clause. Thus, if C' is satisfied, some a_i is true and so C is satisfied.

$\Leftarrow$ If C has a satisfying assignment $\rightarrow$ C' has a satisfying assignment

Assume w.l.o.g that a_i is the first true literal in C. Then, set $y_1, \dots, y_{i-2}$ to <u>true</u> and the rest to <u>false</u>. This satisfies all of C'.

Thus, any instance of SAT can be (poly-time) transformed into an instance of 3-SAT such that the SAT instance has a satisfying assignment iff the 3-SAT instance does. So

$$SAT \leq_p 3\text{-}SAT$$

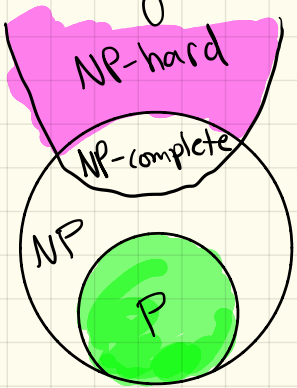
Beyond P, NP and NP-complete.

Note that we have talked about NP-complete problems as decision problems that are at least as hard as any problem in NP.

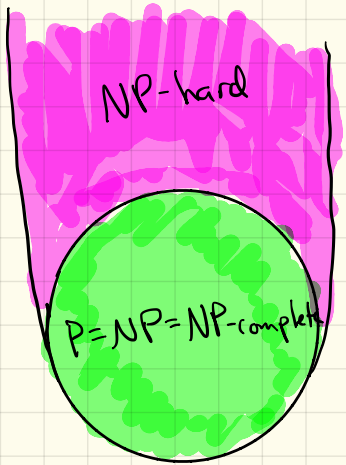
Do not confuse NP-complete with NP-hard.

NP-complete problems are required to be in NP (to have efficient certifiers) NP-hard problems are not, and some NP-hard problems are not decidable in any finite time.

Think of the following sketch:



If $P \neq NP$



If $P = NP$

Some NP-hard problems have decision versions that are NP-complete.

For example, the Traveling Salesman Problem asks to find the shortest Hamiltonian cycle (a cycle that visits each vertex exactly once, except the start vertex to which it returns) in a graph.

The decision problem asks if a graph G has a Hamiltonian cycle of length $\leq k$, for which the cycle itself is an efficient certificate.

Clearly the decision version of TSP is in NP.

However, some decision problems are NP-hard but <u>not</u> NP-complete (because they are not in NP).

One famous example is the <u>halting problem</u>.

Informally the halting problem asks us to determine, for pairs (i, x) of programs and input, will program i halt (eventually terminate) on input x.

We wish to know if there exists an algorithm (Turing Machine) that can solve the halting problem in <u>finite</u> time.

Unfortunately, there is not. The proof itself is a bit involved, but here is the basic concept.

Assume that we had some algorithm, halts(i, x)
TRUE if program i halts and FALSE if it does not.

Now, recognize that a program (i) is just data, so we could write the following function

```
def SelfHalt(p){
    return halts(p,p)
}
```

That is, does program p halt when run on itself?

Finally, consider feeding the following program p to SelfHalt()

```
def paradox(prog){
    if SelfHalt(prog){
        while (True){}
        return False;
    } else {
        return True
    }
}
```

Now, imagine calling SelfHalt(paradox). This will run the paradox() function on itself.

If paradox(paradox) halts, then SelfHalt(paradox) should be True, but in that case we take the if branch, and the call goes into an infinite loop.

If paradox(paradox) does not halt, then we go into the else branch and return True ... which is wrong because SelfHalt(paradox) should be False.

This means that such a general algo as halts() can not exist!

So, the halting problem is undecidable. There is no way to solve it on all instances in finite (let alone polynomial) time!

- There are many interesting ideas related to this:

e.g.

Gödel's incompleteness theorem
Implications for static program analysis
Determination of what is even computable (in finite time).

We do not have the time to go deeply into these topics, but I highly recommend you explore them a bit.

The strange case of co-NP

Our definition of NP is fundamentally asymmetric.

An input string is a "yes" instance iff $\exists\ t$ with $|t| = p(|s|)$ so that $B(s,t) =$ "yes". Negating this statement, a string s is a "no" instance iff <u>for all</u> t', $B(s,t') =$ no. That is:

- It is <u>easy</u> to verify we have a solution
- It is <u>hard</u> to verify that <u>no</u> solution exists

NP is concerned with the efficient verification of yes instances.

For every problem X, the complementary problem $\overline{X}$ is defined so that for all inputs s:

$$s \in \overline{X} \quad \text{iff} \quad s \notin X$$

Note: if $X \in P$ then $\overline{X} \in P$, but this is <u>not</u> necessarily the case for NP.

Def: co-NP - A problem belongs to co-NP if its complementary problem belongs to NP.

Example: Subset Sum $\in$ NP. It asks, given a collection of numbers, is there a non-empty subset whose sum is 0.

Consider the complement: Given a collection C of numbers, return True iff no non-empty subset of C sums to 0. How does one efficiently verify that no such subset exists? I can efficiently verify "no" instances of this problem using the same certificate as for subset sum, but how do I verify "yes" instances?

Open Question:

$$NP = co\text{-}NP \ ?$$

This is widely believed not to be the case. In fact

If $NP \neq co\text{-}NP$ then $P \neq NP$

If $NP \neq co\text{-}NP$ then $P \neq NP$

Proof (contrapositive): $P = NP \Rightarrow NP = co\text{-}NP$

P is closed under complementation, so if $P = NP$, then NP is closed under complementation as well. Specifically, assume $P = NP$:

$$x \in NP \Rightarrow x \in P \Rightarrow \bar{x} \in P \Rightarrow \bar{x} \in NP \Rightarrow x \in co\text{-}NP$$

and

$$x \in co\text{-}NP \Rightarrow \bar{x} \in NP \Rightarrow \bar{x} \in P \Rightarrow x \in P \Rightarrow x \in NP$$

Hence $NP \subseteq co\text{-}NP$ and $co\text{-}NP \subseteq NP$ so $NP = co\text{-}NP$ ∎

The set of <u>decidable</u> problems is called R.

Let's end on a somewhat disturbing note:

Almost all decision problems $\notin \mathbb{R}$

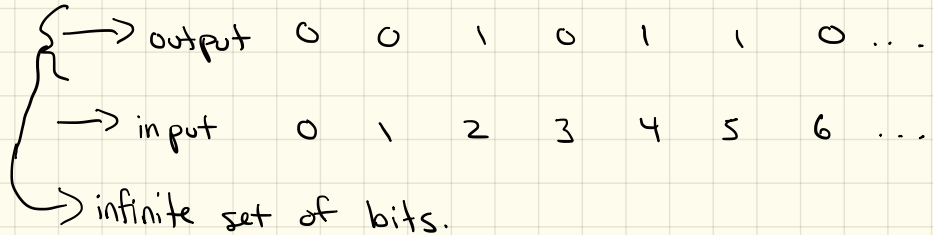
Why (proof concept from E. Demaine)

Consider the following sets

- Turing Machines
  $\approx$
  Binary strings
  $\approx$
  Natural Numbers
  (countably infinite) $\mathbb{N}$

- Decision problems
  = function from input $\to \{0,1\}$
  input $\approx$ binary string $\approx$ natural number $\approx \mathbb{N}$

  so it's a function from $\mathbb{N} \to \{0,1\}$



# of functions $\mathbb{N} \to \{0,1\}$ has a cardinality much greater than $\mathbb{N}$. It is uncountably infinite.

Almost all problems have no algorithm that solve them!