

NP-completeness

- The goal of this section is to identify the “hardest” problems in **NP**
- We’ll consider a language B to be “harder” than (or more accurately, “at least as hard as”) a language A if a polynomial-time algorithm for B could be used to decide A in polynomial time
- **Definition:** Let A and B be languages. Then $A \leq_p B$ (“ A reduces to B in polynomial time”) if there exists a function f that is computable in polynomial time, such that for all $x \in \Sigma^*$:

$$x \in A \iff f(x) \in B$$

Note that a function f is computable in polynomial time if there exists a Turing machine M and a polynomial q such that $T_M(n) \leq q(n)$ for all n and for every $x \in \Sigma^*$, M on input x outputs $f(x)$.

- **Fact:** If $A \leq_p B$ and $B \in \mathbf{P}$ then $A \in \mathbf{P}$.
- **Proof:** Let f be a function, computable in polynomial time, reducing A to B , and let M_B be a polynomial time Turing machine deciding B . Then the following Turing machine M_A decides A in polynomial time: M_A = “on input x ...

1. Compute $y = f(x)$
2. Run M_B on input y , and accept iff M_B accepts

Let q_1 be a polynomial such that f is computable in time $O(q_1(n))$. Then the first step takes time $O(q_1(|x|))$. Also, $f(x)$ cannot be longer than $q_1(|x|)$. Let q_2 be a polynomial such that $T_{M_B}(n) \leq q_2(n)$. Then the second step takes time $O(q_2(|f(x)|)) = O(q_2(q_1(x)))$. Since q_1 and q_2 are polynomials, so is the composition of q_1 and q_2 .

- **Definition:** A language L is **NP-hard** if for every language $L' \in \mathbf{NP}$, $L' \leq_p L$. A language L is **NP-complete** if the following two conditions hold:

1. $L \in \mathbf{NP}$
2. L is **NP-hard**

- Notice that if a language is **NP-hard**, then a polynomial-time algorithm for that language could be used to decide *every* language in **NP** in polynomial time! In fact, the following theorem shows that to study the question of whether $\mathbf{P} = \mathbf{NP}$, it suffices to establish the complexity of a single **NP-complete** problem.

- **Theorem:** Let L be **NP-complete**. Then $L \in \mathbf{P} \iff \mathbf{P} = \mathbf{NP}$.

- **Proof:** Let L be **NP-complete**.

- (\Rightarrow) Suppose $L \in \mathbf{P}$. Then for every language $L' \in \mathbf{NP}$, we have that $L' \leq_p L$ (since L is \mathbf{NP} -hard), and as $L' \leq_p L$ and $L \in \mathbf{P}$, it follows that $L' \in \mathbf{P}$. Thus every language in \mathbf{NP} is also in \mathbf{P} , or in other words $\mathbf{NP} \subseteq \mathbf{P}$. Since we already know that $\mathbf{P} \subseteq \mathbf{NP}$, it follows that $\mathbf{P} = \mathbf{NP}$.
- (\Leftarrow) Suppose that $\mathbf{P} = \mathbf{NP}$. Then $L \in \mathbf{NP} = \mathbf{P}$, since L is \mathbf{NP} -complete.
- So if we find a polynomial-time algorithm for *one* \mathbf{NP} -complete language, then $\mathbf{P} = \mathbf{NP}$. On the other hand, if we can prove that there does not exist a polynomial-time algorithm for *one* \mathbf{NP} -complete language, then $\mathbf{P} \neq \mathbf{NP}$.

Proving NP-hardness

- **Fact:** If $L_1 \leq_p L_2$ and $L_2 \leq_p L_3$ then $L_1 \leq_p L_3$ (this property is called “transitivity”).
- **Proof:** Let $f_{1,2}$ be a polytime reduction from L_1 to L_2 , and let $f_{2,3}$ be a polytime reduction from L_2 to L_3 . Define

$$f_{1,3}(x) = f_{2,3}(f_{1,2}(x))$$

Then $f_{1,3}$ is computable in polynomial time, as polynomials are closed under composition, and

$$\begin{aligned} x \in L_1 &\iff f_{1,2}(x) \in L_2 \\ &\iff f_{2,3}(f_{1,2}(x)) \in L_3 \\ &\iff f_{1,3}(x) \in L_3 \end{aligned}$$

So $f_{1,3}$ is a polytime reduction from L_1 to L_3 .

- **Fact:** If $L' \leq_p L$ and L' is \mathbf{NP} -hard, then L is \mathbf{NP} -hard.
- **Proof:** We need to show that for every language $L'' \in \mathbf{NP}$, $L'' \leq_p L$. Let $L'' \in \mathbf{NP}$. Then $L'' \leq_p L'$, as L' is \mathbf{NP} -hard, and since $L' \leq_p L$ it follows by transitivity that $L'' \leq_p L$. So every language in \mathbf{NP} reduces to L in polynomial time.
- So to prove that a language is \mathbf{NP} -hard, it suffices to show that some \mathbf{NP} -hard language reduces to it in polynomial time.

NP-complete languages

The canonical NP-complete language

- **Definition:**

$$L_{\mathbf{NP}} = \{ \langle M, w, 1^t \rangle \mid M \text{ is an NTM that has an accepting computation of length at most } t \text{ on input } w \}$$

- **Theorem:** L_{NP} is **NP**-complete.
- Proof: Homework.

Satisfiability of boolean formulae

- A boolean formula consists of the connectives \wedge (AND), \vee (OR), and \neg (NOT); variables x_1, x_2, x_3, \dots ; and parentheses
- A truth assignment τ for a formula φ maps each variable in φ to $\{0, 1\}$
- Applying τ to φ in the natural way gives a truth value $\varphi(\tau) \in \{0, 1\}$ for the whole formula
- A formula φ is satisfiable if there exists a truth assignment τ such that $\varphi(\tau) = 1$.
- **Definition:** $\text{SAT} = \{\langle \varphi \rangle \mid \varphi \text{ is a satisfiable boolean formula}\}$
- **Theorem:** SAT is **NP**-complete
- Proof: We must show two things: first, that $\text{SAT} \in \text{NP}$, and second, that SAT is **NP**-hard.

- SAT \in **NP**: Describe a “guess and check” algorithm for SAT. Define $M =$ “on input $\langle \varphi \rangle$...”
 1. Guess a truth assignment τ for φ
 2. Accept iff $\varphi(\tau) = 1$

We must prove that $L(M) = \text{SAT}$. If $\langle \varphi \rangle \in L(M)$ then M has an accepting computation on input $\langle \varphi \rangle$. Since M accepts only if it discovers a satisfying truth assignment, it follows that $\langle \varphi \rangle \in \text{SAT}$. On the other hand, if $\langle \varphi \rangle \in \text{SAT}$ then φ is satisfiable, so M has an accepting computation on input $\langle \varphi \rangle$, in which it correctly guesses a satisfying assignment. So we have shown that $\langle \varphi \rangle \in L(M) \iff \langle \varphi \rangle \in \text{SAT}$, implying that $L(M) = \text{SAT}$.

What is the running time of M ? Guessing a truth assignment takes only linear time, since the number of variables is no larger than the size of the formula. To evaluate $\varphi(\tau)$, M can begin by replacing every variable x_i with $\tau(x_i)$, and then iteratively search for subexpressions like $(b_1 \vee b_2)$, $(b_1 \wedge b_2)$, or $\neg(b_1)$, where $b_1, b_2 \in \{0, 1\}$, and replace these expressions with their values. One such iteration takes time $O(|\varphi|)$, and as each iteration decreases the size of the formula there can be at most $|\varphi|$ iterations, so the time required is $O(|\varphi|^2)$.

- SAT is **NP**-hard: next class