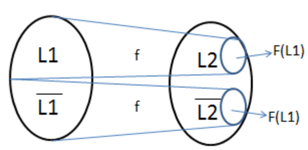
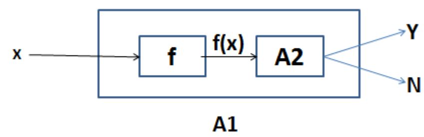
# Advanced Algorithms and Complexity : Lecture 3 Polynomial Time Reductions. The Complexity Classes NP-Complete and NP-Hard. The Satisfiability Problem.

August 8, 2018

**Polynomial Time Reductions:** We say that a language $L_1$ reduces to a language $L_2$ in polynomial-time (denoted as $L1 \leq_p L_2$) if there exists a polynomial-time computable function $f(x)$ (there exists a DTM which takes as inputs a string $x$ and gives as output the string $f(x)$) such that $x \in L_1 \implies f(x) \in L_2$.



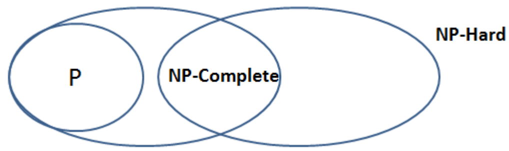

If $L_1 \leq_p L_2$ and $L_2$ has a polynomial-time algorithm $A_2$, then we can combine $A_2$ and $f$ to get a polynomial time algorithm $A_1$ for $L_1$ as follows:




First $x$ is given as input to the DTM for computing $f(x)$ in polynomial-time. Then $f(x)$ is given as input to DTM $A_2$. If $A_2$ accepts $f(x)$, then $A_1$ accepts $x$. If $A_2$ rejects $f(x)$, then $A_1$ rejects $x$. The total time taken is polynomial since both DTM's take polynomial-time. The DTM $A_1$ accepts $L_1$ because $x \in L_1 \iff f(x) \in L_2$.
Polynomial-time reductions are transitive: If $L_1 \leq_p L_2$, and $L_2 \leq_p L_3$, then $L_1 \leq_p L_3$.
$L_1 \leq_p L_2 \implies \exists$ a polynomial-time DTM computing $f$ such that $x \in L_1 \iff f(x) \in L_2$.
$L_2 \leq_p L_3 \implies \exists$ a polynomial-time DTM computing $g$ such that $y \in L_2 \iff g(y) \in L_3$.
Putting $y = f(x)$, we get: $x \in L_1 \iff f(x) \in L_2 \iff g(f(x)) \in L_3$. Since $f(x)$ can be computed in polynomial-time, $|f(x)|$ is of polynomial-length and also because $g(y)$ can be computed in polynomial-time, $(gof)(x)$ can be computed in polynomial time $\implies L_1 \leq_p L_3$.

**NP-Complete:** A language $L_1 \in NP$ is called completed for $NP$, or NP-complete if $\forall L \in NP$, $L \leq_p L_1$.

**NP-Hard:** A Language $L_2$ is called hard for $NP$, or NP-Hard if $\forall L \in NP$, $L \leq_p L_2$.
The definitions of NP-complete and NP-Hard are similar, with only one difference: NP-Complete language should belong to $NP$, but NP-Hard Language may not be in $NP$.
An NP-Complete Language is always NP-Hard, but an NP-Hard Language may not be NP-Complete. It is unknown whether $P = NP$, or $P \neq NP$. Usually it is believed that $P \neq NP$. Earlier we have seen that $P \subseteq NP$. We also have the relations NP-Complete $\subseteq NP$, and NP-Complete $\not\subseteq$ NP-Hard. One possibility is:




An NP-Complete Language is called complete for $NP$ because a polynomial-time algorithm for the NP-complete language can be combined with the polynomial-time reduction to get a polynomial-time algorithm for any problem in $NP$ (as we have seen earlier):
$L \in$ NP-complete and $L \in P \implies P = NP$.
An NP-Hard Language is called hard for $NP$ bacause it may be "harder" than any problem in $NP$ (it doesn't belong to NP-complete). As before, if we are able to find a polynomial-time algorithm for an NP-Hard problem, then we can combine it with the polynomial-time reduction to get a polynomial-time algorithm for any problem in $NP$:
$L \in$ NP-Hard and $L \in P \implies P = NP$.

**A Boolean Formula** over the variables $u_1, u_2, ..., u_n$ consists of the variables and the logical operators AND ($\wedge$), OR ($\vee$) and NOT($\neg$).
Example: $(u_1 \wedge u_2) \vee (u_2 \wedge u_3) \vee (u_3 \wedge u_1)$. If $\Phi$ is a Boolean formula over variables $u_1, u_2, ..., u_n$, and $z \in \{0, 1\}^n$, then $\Phi(z)$ denotes the value of $\Phi$ when the variables of $\Phi$ are assigned the values $z$. $(1 = \text{True}, 0 = \text{False})$. A formula $\Phi$ is satisfiable if there exists some assignment $z$ such that $\Phi$(z) is true. Otherwise, we say that $\Phi$ is unsatisfiable.
Example: The formula $x \wedge^{\neg} x$ ( $= 0$ ) is not satisfiable.
A Boolean formula over variables $u_1, u_2, ..., u_n$ is in CNF form (Conjunctive Normal Form) if it is an AND of OR's of variables or their negation. Example of a 3CNF formula: $(u_1 \vee \overline{u_2} \vee u_3) \wedge (u_2 \vee \overline{u_3} \vee u_4) \wedge (\overline{u_1} \vee u_3 \vee \overline{u_4})$.
More generally, a CNF formula has the form: $\wedge_i(\vee_j v_{ij})$ where each $v_{ij}$ is either a variable $u_k$ or is negation $\overline{u_k}$. The terms $\vee_j v_{ij}$ are called its clauses. A $k$CNF formula is a CNF formula in which all clauses contain at most $k$ literals. We denote by SAT the language of all satisfiable CNF formulae and by 3SAT the language of all satisfiable 3CNF formulae.

**SAT is NP-Complete (Cook-Levin Theorem):** SAT $\in$ NP-Complete can be proved in two steps:

**1. SAT $\in NP$:** Given a Boolean formula $\Phi(z)$ as input, the NTM $N$ will guess an assignment of variables (0 or 1) for $z$, and then it will evaluate $\Phi(z)$ in polynomial time. If $\Phi(z)$ evaluates to 1, then $N$ will accept, oherwise it will reject.

**2. SAT $\in$ NP-Hard:** For this we will have to prove that for any $L \in NP$, $L \leq_p$ SAT. We will have to describe a polynomial-time DTM $M$ such that:
$x \in L \implies M(x) \in$ SAT and
$x \notin L \implies M(x) \notin$ SAT.
$L \in NP \implies \exists$ a polynomial time DTM $D$ such that:
$x \in L \implies \exists y \in \{0,1\}^{p(|x|)}$ such that $D(x,y) = 1$
$x \notin L \implies \forall y \in \{0,1\}^{p(|x|)}$, $D(x,y) = 0$
First (unsuccessful) attempt in designing $M$: $M$ will take input $x$, and it will simulate $D(x,y)$ for all $y \in \{0,1\}^{p(|x|)}$. If for any such $y$ it gets 1 as output, then it will output $x \wedge x$ ($\in$ SAT), otherwise it will output $x \wedge \neg x$ ($\notin$ SAT). $M$ will take exponential-time. $M$ will be a polynomial-time DTM only for the case of $L \in P$ ($|y| = 0$).