

CS 535: Complexity Theory, Fall 2026

Homework 3

Due: 11:59PM, Friday, September 25, 2026.

Reminder. Homework must be typeset with $\text{\LaTeX}$ preferred. Problem 1 (all parts), Problem 2 parts (a)-(d), and Problem 3 parts (a) and (b) are required. Problem 2 part (e) and Problem 3 part (c) are optional.

Problem 1 (*Required*) **Max Independent Set**. Recall that an independent set in an undirected graph $G = (V, E)$ is a set of vertices $S \subseteq V$ such that no pair of vertices in S are connected by an edge. The language

$$\text{INDSET} = \{\langle G, k \rangle \mid G \text{ has an independent set of size } \geq k\}$$

is **NP**-complete.

- (a) Show that if $\mathbf{P} = \mathbf{NP}$, then there is a polynomial-time algorithm that, given an undirected graph G , outputs a maximum-size independent set in G .
- (b) An **NP maximization problem** is specified by a polynomial-time computable function $f : \{0, 1\}^* \times \{0, 1\}^* \rightarrow \mathbb{N}$ (whose output is written in binary) and a polynomial p . On input x , the task is to find a string $y \in \{0, 1\}^{p(|x|)}$ maximizing $f(x, y)$.

Prove that every **NP** maximization problem has a polynomial-time algorithm if and only if $\mathbf{P} = \mathbf{NP}$. Your argument should explain both how to determine the maximum value and how to obtain a maximizing string.

Problem 2 (Reductions to Sparse Languages). A language $L \subseteq \{0, 1\}^*$ is *sparse* if there exists a polynomial $p(n)$ such that $|L \cap \{0, 1\}^{\leq n}| \leq p(n)$ for every natural number n . That is, for every n , out of the $2^{n+1} - 1$ possible strings of length $\leq n$, only polynomially many (a tiny fraction) are in L . In this problem, you will prove and explore some consequences of *Fortune's Theorem*, which says that the existence of a **coNP**-complete sparse language implies $\mathbf{P} = \mathbf{NP}$.

So let's get started! Suppose there exists a **coNP**-complete language L such that there is a polynomial p such that $|L \cap \{0, 1\}^{\leq n}| \leq p(n)$ for every n . Consider the **coNP**-complete language $\text{TAUT} = \{\varphi \mid \varphi \text{ is a Boolean formula s.t. } \forall x \varphi(x) = 1\}$. Since L is **coNP**-complete, there is a poly-time reduction f from TAUT to L , and therefore some polynomial r such that $|f(\varphi)| \leq r(|\varphi|)$ for every formula φ . To prove Fortune's Theorem, it suffices to exhibit a poly-time algorithm for TAUT .

To give such an algorithm, first consider (as a thought experiment) building the downward self-reduction tree for an input formula. That is, given a Boolean formula $\varphi(x_1, \dots, x_m)$, let $\varphi_0 = \varphi(0, x_2, \dots, x_m)$ and $\varphi_1 = \varphi(1, x_2, \dots, x_m)$. Then $\varphi \in \text{TAUT} \iff \varphi_0 \in \text{TAUT} \text{ and } \varphi_1 \in \text{TAUT}$. We can recurse on the formulas φ_0 and φ_1 , ultimately giving us a tree of depth m

such that the original formula is a tautology iff all the leaves evaluate to 1, but this takes exponential time as there are 2^m leaves.

Instead, we're going to prune the tree as we explore it, ensuring that the number of formulas explored at each depth is bounded by a polynomial. We will take this polynomial to be $t(n) := p(r(n))$, where $n = |\varphi|$, for reasons that will hopefully become clear. Suppose that at some level of the tree, we have a collection of “active” formulas $\varphi_1, \dots, \varphi_k$ such that $\varphi \in \text{TAUT} \iff \varphi_i \in \text{TAUT}$ for all $i = 1, \dots, k$. If $k \leq t(n)$, then we recurse on to the next level of the tree. Otherwise, if $k > t(n)$, we will prune the set of active formulas by one.

Here's how to do the pruning. For each $i = 1, \dots, k$, let $s_i = f(\varphi_i)$ be the result of applying the reduction f to the formula φ_i . So if every active formula $\varphi_1, \dots, \varphi_k$ has length at most n , then every string $s_1, \dots, s_k$ has length at most $r(n)$.

Examining the collection of resulting strings $s_1, \dots, s_k$, there are two possible cases:

Case 1: All strings $s_1, \dots, s_k$ are distinct.

Case 2: There exists a pair of strings s_i, s_j where $s_i = s_j$ but $i \neq j$.

- (a) Show that in Case 1, we can automatically conclude that φ is NOT a tautology, and therefore we can halt and reject.
- (b) Show that in Case 2, it is safe to prune, say, the formula φ_i . That is, the original formula φ is a tautology if and only if $\varphi_1, \dots, \varphi_{i-1}, \varphi_{i+1}, \dots, \varphi_k$ are all tautologies.

Thus, if $k > t(n)$, our algorithm either halts or reduces the number of active formulas by 1.

- (c) Briefly analyze the runtime of this algorithm to conclude that it indeed decides TAUT in polynomial time.
- (d) A natural question you might ask about TAUT (or SAT or whatever) is whether there is a poly-time algorithm that correctly solves it on *most* instances. More specifically, let's say that an algorithm A *almost solves* TAUT (with one-sided error) if:
 - φ is not a tautology $\implies A(\varphi) = 0$. (I.e., A always produces the correct answer of 0 whenever φ is not a tautology.)
 - There exists a polynomial $p(n)$ such that for every n , we have $A(\varphi) = 0$ for at most $p(n)$ tautologies φ of length at most n . (I.e., A produces the correct answer of 1 for all but polynomially many tautologies φ .)

Use Fortune's Theorem to show that if there is a deterministic poly-time algorithm that almost solves TAUT with one-sided error, then $\mathbf{P} = \mathbf{NP}$.

- (e) (*Optional*) Show that the conclusion above holds even for algorithms A with two-sided error. That is, if there is a polynomial $p(n)$ and a deterministic poly-time algorithm A such that for every n ,
 - $A(\varphi) = 0$ for at most $p(n)$ tautologies φ of length n , and

- $A(\varphi) = 1$ for at most $p(n)$ non-tautologies φ of length n ,

then $\mathbf{P} = \mathbf{NP}$.

Problem 3 (Time Hierarchies). In this problem, you may assume that any reasonable-looking function is time-constructible. Technically speaking, parts (a) and (b) follow from part (c), so you could just solve that problem. But it will be very helpful to solve these problems in the order specified.

- (a) Prove that

$$\mathbf{DTIME}(n) \subsetneq \mathbf{DTIME}(n \log^2 n).$$

- (b) Prove that

$$\mathbf{DTIME}(n) \subsetneq \mathbf{DTIME}(n \log n).$$

Hint: Use padding.

- (c) (*Optional*) Prove that for every constant $\varepsilon > 0$,

$$\mathbf{DTIME}(n) \subsetneq \mathbf{DTIME}(n \log^\varepsilon n).$$