Problem 1 (Decision vs. Optimization). An NP minimization problem is specified by a polynomial-time computable objective function $f : \{0,1\}^* \times \{0,1\}^* \rightarrow \mathbb{N}$ and a polynomial $p$. Given an input $x \in \{0,1\}^*$, the problem is to find a $y \in \{0,1\}^{p(|x|)}$ that minimizes $f(x, y)$, i.e., find a string in $\arg\min_{y \in \{0,1\}^{p(|x|)}} f(x, y)$.

(a) Given a collection of sets $S_1, \ldots, S_m \subseteq [n]$, a hitting set is a set $T$ such that $T \cap S_i \neq \emptyset$ for every $i = 1, \ldots, m$. Describe how the task of finding a minimum-size hitting set can be stated as an NP-minimization problem. (2 points)

(b) Consider the decision problem $HS = \{\langle n, k, S_1, \ldots, S_m \rangle \mid \exists$ a hitting set for $S_1, \ldots, S_m$ of size $\leq k\}$. Show that if $HS \in P$, then there is a poly-time algorithm for finding a minimum-size hitting set. (3 points)

(c) Show that $P = \text{NP}$ if and only if every NP minimization problem can be solved in polynomial time. (4 points)

Hint: We showed in class that $P = \text{NP}$ iff every NP search problem can be solved in poly-time. You can use this fact without proof.

Problem 2 (Sparse Languages). A language $L \in \{0,1\}^*$ is sparse if there exists a polynomial $p(n)$ such that $|L \cap \{0,1\}^n| \leq p(n)$ 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 $P = \text{NP}$.

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

(a) Briefly explain why, in order to prove Fortune’s Theorem, it suffices to exhibit a poly-time algorithm for $\text{TAUT}$. (1 point)
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 $\varphi(x_1, \ldots, x_n)$, let $\varphi_0 = \varphi(0, x_2, \ldots, x_n)$ and $\varphi_1 = \varphi(1, x_2, \ldots, x_n)$. Then $\varphi \in \text{TAUT} \iff \varphi_0 \in \text{TAUT}$ and $\varphi_1 \in \text{TAUT}$. We can recurse on the formulas $\varphi_0$ and $\varphi_1$, ultimately giving us a tree of depth $n$ such that the original formula is a tautology iff all the leaves evaluate to 1, but this takes exponential time as there are $2^n$ 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 (roughly) $t(n) := p(r(2n + 5))$, for reasons that will hopefully become clear. Suppose that at some level of the tree, we’ve materialized a collection of “active” formulas $\varphi_0, \ldots, \varphi_k$ such that $\varphi \in \text{TAUT} \iff \varphi_i \in \text{TAUT}$ for all $i = 0, 1, \ldots, k$. 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 $i = 1, \ldots, k$ (note that we are not including $i = 0$), let $s_i = f((\varphi_0) \land (\varphi_i))$. That is, we apply the reduction $f$ to the formula obtained by taking the logical AND of formulas $\varphi_0$ and $\varphi_i$.

(b) Explain why if every active formula $\varphi_0, \ldots, \varphi_k$ has length at most $n$, then every string $s_1, \ldots, s_k$ has length at most $r(2n + 5)$. (1 point)

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

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

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

(c) Show that in Case 1, we can automatically conclude that $\varphi$ is NOT a tautology, and therefore we can halt and reject. (2 points)

(d) Show that in Case 2, it is safe to prune, say, the formula $\varphi_i$. That is, the original formula $\varphi$ is a tautology if and only if $\varphi_0, \ldots, \varphi_{i-1}, \varphi_{i+1}, \ldots, \varphi_k$ are all tautologies. (2 points)

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

(e) Briefly analyze the runtime of this algorithm to conclude that it indeed decides \text{TAUT} in polynomial time. (2 points)

(f) A natural question you might ask about \text{TAUT} (or \text{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 $A$ \textit{almost solves} \text{TAUT} (with one-sided error) if:

- $\varphi$ is not a tautology $\implies A(\varphi) = 0$. (I.e., $A$ always produces the correct answer of 0 whenever $\varphi$ 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 $\varphi$ of length at most $n$. (I.e., $A$ produces the correct answer of 1 for all but polynomially many tautologies $\varphi$.)

Use Fortune’s Theorem to show that if there is a poly-time algorithm that almost solves $\textsc{TAUT}$ with one-sided error, then $P = \textbf{NP}$. (3 points)

(g) (*Bonus*) 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 poly-time algorithm $A$ such that for every $n$,

- $A(\varphi) = 0$ for at most $p(n)$ tautologies $\varphi$ of length $n$, and
- $A(\varphi) = 1$ for at most $p(n)$ non-tautologies $\varphi$ of length $n$,

then $P = \textbf{NP}$. 