Problem 1

Prove the general case of the deterministic time hierarchy theorem. We have a TM D that takes input x, and runs $\mathcal{U}$ (with logarithmic slowdown) for a limited number of steps to simulate M_x on x. If $\mathcal{U}$ outputs a bit b, then D outputs the opposite bit $(1-b)$, and otherwise outputs 0.

By the statement of the Time Hierarchy Theorem, we have functions f and g satisfying $f(n)\log f(n) = o(g(n))$. If we limit the number of steps that $\mathcal{U}$ may take to exactly $g(|x|)$, then the language L decided by D is by definition $\mathbf{DTIME}(g(|x|))$. We claim that $L \notin \mathbf{DTIME}(f(|x|))$. Suppose for the sake of contradiction that there were a TM M and constant c_1 such that on input $x \in \{0,1\}^\star$ M outputs $D(x)$ in $c_1 \cdot f(|x|)$ steps.

The time to simulate M by $\mathcal{U}$ on every input x is at most $c_2 c_1 f(|x|)\log f(|x|)$ for some constant c_2 that depends only on the size of the alphabet, the number of tapes, and the number of states of M, and not on x. By the definition of $o(\cdot)$, there is some integer n_0 such that $g(n_0) > c_2 c_1 f(n_0)\log f(n_0)$ for all $n \geq n_0$. Since every TM can be represented by infinitely many strings, there is a string x that encodes M and is of length at least n_0, such that even if a smaller encoding of M doesn't halt before time runs out, a larger equivalent encoding will. $D(x)$ will then output $b = M(x)$ within at most $g(|x|)$ steps, but by the definition of D, $D(x) = 1 - b \neq M(x)$, giving the contradiction.

Problem 2

Prove the definition of $H(n)$ in Ladner's theorem implies an $O(n^3)$-time algorithm to compute $H(n)$ from n. To compute $H(n)$ from n by brute force, we just need to iterate through the possible values of i up to $\log\log n$, and for each one simulate the TM M_i for $i|x|^i$ steps on every x in $\{0,1\}^\star$ with $|x| \leq \log n$, and for each x verify that the output is $\mathsf{SAT}_H(x)$, which requires recursively computing $H(|x|)$ and solving the instance of SAT encoded in x. If we find an i for which all of this works out, then we're done, and if we get through $\log\log n$ without finding a suitable i, then $H(n)$ is just $\log logn$.

In the worst case we need to compute $H(i)$ for each $i \leq \log n$ (to know how many 1s to append to instances of $\mathsf{SAT_H}$), simulate $\log\log n$ machines on inputs of length at most $\log n$ (that's $O(2^{\log n}) = O(n)$ inputs total) for a maximum of $\log\log n(\log n)^{\log\log n} = o(n)$ steps, and compute SAT on $O(\log\log n \cdot n)$ formulas of size at most $\log n$. Solving an instance of SAT on inputs of size $\log n$ takes just $O(n)$ time, so if the time to compute $H(n)$ is $T(n)$, $T(n) \leq \log nT(\log n) + O(\log\log n \cdot n \cdot (n+n)) = \log nT(\log n) + O(n^3) = O(n^3)$.

Problem 3

Prove that SAT^A is $\mathbf{NP}^A$-complete. SAT^A is in $\mathbf{NP}^A$; as with SAT, the variable assignment will serve as a certificate. To show that SAT^A is $\mathbf{NP}$-hard, we use a modified version of the Cook-Levin reduction. The idea is to reduce the problem of deciding any language $L \in \mathbf{NP}^A$ to the problem of deciding SAT^A by transforming a TM M^A that decides L and an input x into an instance of SAT^A.

We need to modify the Cook-Levin reduction to account for the special oracle tape that the verifier M^A has access to. We can encode the oracle tape on the single work/output tape of the oblivious

TM, which is okay because we still know exactly where the $T(n)$ cells of the oracle tape reside. The snapshots remain the same, but the function F that verifies that snapshot z_i follows from the previous snapshot, the current bit of input, and the snapshot from the last time that M was at the same position needs to be modified to deal with the oracle. The function must now take as an extra argument the contents of the oracle tape, which will be used to verify that the next state is q_{no} or q_{yes} if the current state is q_{query}. The extra argument can be translated to an A clause from the definition of SAT^A, so that although it will depend on the entire $T(n)$ cells of the oracle tape, it will add a maximum of $T(n)^2$ variables to the final formula ϕ_x, keeping the size of the formula polynomial. We have access to the contents of the oracle tape via the snapshots, which encode the symbol on each cell of the query tape the last time it was visited before snapshot z_i. This process requires running through at most $T(n)$ snapshots $T(n)$ times, so the time to build the formula is still polynomial.

Show there exists an A such that SAT^A is not in $\mathbf{P}^A$. We can use the same construction used by Baker, Gill, and Solovay to show that there is an oracle B such that their language $\mathsf{U}_B \notin \mathbf{P}^B$. We can do this because $\mathsf{U}_B \subsetneq \mathsf{SAT}^A$. Any instance of U_B is just an instance of SAT^A with one A clause and no other restrictions on the variables $x_1 \dots x_n$. To construct the language A, we enumerate every oracle TM, and ensure that each one cannot decide instances of SAT^A with just one A clause of any size in polynomial time. We build the language A in phases, starting with A clauses of size 1, then 2, and so on. We do this by simulating, for every i, each M_i on an input of a single A clause for up to $2^n/10$ steps. For each machine that we simulate, we choose an A clause for the input of size larger than any A clause we've already ruled out, and make sure that whatever M_i decides, it is wrong. If M_i queries whether some string is in A we answer consistently for strings we've already determined, and "no" otherwise. If M_i accepts some input, we remove all inputs of that size from A. Conversely, if M_i rejects some input, we pick an input of that size that we haven't already rejected, and add it to A. Thus we guarantee that no A clause can be decided in polynomial time, and consequently SAT^A can't be decided in polynomial time either.

Problem 4

Show that $\mathsf{2SAT}$ *is in* $\mathbf{NL}$. The proof relies on the observation that the clauses of an instance of $\mathsf{2SAT}$ are implications. For example, the clause $a \vee b \equiv \neg a \implies b$, and the clause $\neg a \vee b \equiv a \implies b$. Each implication can be thought of as an edge in a graph from a vertex labeled by the first variable to a vertex labeled by the second (where $\neg x$ and x are different vertices). The conjunction of each of these implications describes a graph, and if there is a path in that graph from x to $\neg x$ and from $\neg x$ back to x (for any variable x), then the formula is unsatisfiable because such paths are equivalent to the unsatisfiable formula $x \iff \neg x$.

We can use the proof of the Immerman-Szelepcsényi theorem that decides $\overline{\mathsf{PATH}}$ in log space to confirm that for each variable in an instance of $\mathsf{2SAT}$, there is no path from that variable to its negation and back. If for all variables there are no such paths, then the formula is satisfiable, and otherwise it is not. We define an NDTM M that takes as input x, an instance of $\mathsf{2SAT}$, and outputs 1 if the formula is satisfiable, and 0 otherwise.

M can convert any clause of a $\mathsf{2SAT}$ instance into an implication $x \implies y$ in log space, and can guess another clause that converts to the implication $y \implies z$. In this way, M can use its nondeterminism to guess a path from some variable x to $\neg x$, and to go in the other direction. We need to check each variable for a path to its negation and back, but again this is fine because we can reuse space between checking each pair of paths for a given variable, and we can stop and reject the formula if there ever is such a pair of paths, and otherwise accept once we've checked all variables. We only need to remember which variable we're checking at any given time, which we can do in log space.

Problem 5

Show that TQBF_k *is* $\mathbf{\Sigma}_k^p$*-complete under logspace reductions.* First, $\mathsf{TQBF}_k \in \mathbf{\Sigma}_k^p$ because its k alternating quantifiers with outermost $\exists$ correspond to the quantifiers of the $\mathbf{\Sigma}_k^p$ definition, and the polynomial

TM M just needs to verify that the ϕ of the TQBF_k instance is true for the assignment of variables given by $u_1, \ldots, u_k$. To show that TQBF_k is $\mathbf{\Sigma}_k^p$-complete, we use the Cook-Levin reduction on the TM M from the definition of $\mathbf{\Sigma}_k^p$. The reduction produces a polynomial-sized formula, so we just need to ensure that the reduction can be done in log space.

The main concern is calculating $\mathsf{inputpos}(i)$ and $\mathsf{prev}(i)$, both of which require simulating the oblivious TM M in the original reduction. We don't have the space to do the usual simulation, but because M is oblivious, and we only need to track where the input head is, and the last step before some step i on which the work/output tape head was at a certain position, we don't actually need to store the contents of all of the tapes. Instead, we can just store some counters to track where the input and work/output heads are, and which step we're on, all of which are logarithmic in some polynomial of the input to M. Writing the parts of the boolean formula that verify that the input in the boolean function matches the input to M, that we start in the start state, and that we end in the end state can all be done by simple copying from M and its input, and certainly in log space.

Show TQBF *is* **PSPACE***-complete under logspace reductions.* We can follow the proof from the book that TQBF is **PSPACE**-complete. We have the same concern as with the Cook-Levin reduction, because the construction of the function $\phi_M(C, C')$, which checks that two configurations are adjacent in the configuration graph of M, depends upon $\mathsf{inputpos}(i)$ and $\mathsf{prev}(i)$ as well. We can use the same technique as above to calculate these values in log space. The size of the final formula is $O(m^2)$, but that's fine because it's polynomial, and we can calculate it one bit at a time.