1) To show L is in NP, we will construct a polynomial-time verifier that checks if $y \in L$ gives a witness. The witness for input y is the unique string $x \mid f(x)=y$. Given an input y and witness x, the verifier checks $f(x)=y$ and $g(x)=1$. These can both be done efficiently as f and g are computable in polynomial time. If $y \in L$, then $g(f^{-1}(y))=1$. Since $x=f^{-1}(y)$ is a valid witness, the verifier will accept y. If $y \notin L$, then for any x such that $f(x)=y$, we get $g(x)=0$. Thus, no valid witness exists that meets both conditions, so the verifier rejects. Since the verifier runs in polynomial time given witness x, $L \in NP$. To show L is in coNP, we will provide a polynomial-time verifier to show $y \notin L$. The witness for an input y is the unique string x such that $f(x)=y$. Given an input y and witness x, the verifier checks that $f(x)=y$ and $g(x)=0$, which can both be done in polynomial time. If $y \notin L$, then $g(f^{-1}(y))=0$, so $x=f^{-1}(y)$ is a valid witness that proves $y \notin L$. If $y \in L$, then for any possible x such that $f(x)=y$, we have $g(x)=1$, so no valid witness x exists with $g(x)=0$, so the verifier will reject. Since the verifier runs in polynomial time given witness x, $L \in$ coNP. Since $L \in NP$ and $L \in$ coNP, $L \in NP \cap$ coNP.

2) a) Given a state $v=(v_1, v_2 \dots v_n)$ that is in deadlock, we can check all the pairs i, j to see that $(v_i, v_i'), (v_j, v_j')$ is not in P. We know there are n^2 pairs to check and our verifier traverses through P to check each pair such that the verifier runs in polynomial time. ∴ DEADLOCK is in NP. We will show that DEADLOCK is NP-hard by reduction from 3-SAT. Given an instance of 3-SAT with n variables and m clauses, we create m directed graphs $G_1, G_2, \dots G_m$. one for each clause in the 3-SAT formula. Each graph $G_i = (V_i, E_i)$ is a directed cycle on 3 vertices. The vertices are labelled with the literals appearing in clause i. Our list P contains all pairs of edges $(e_i = (v_i, v_i'), e_j = (v_j, v_j'))$ where $e_i \in E_i$ and $e_j \in E_j$. we ensure that v_i and v_j are negations of each other. This reduction runs in polynomial time. Given a YES instance of 3SAT, we will create a YES instance of DEADLOCK. Consider the instance of DEADLOCK with P and $G_1, G_2, \dots G_m$ with every v_i labelled with true literals such that no two literals negate each other Thus, no such $v_i, v_j \in V$ exists where there exists a pair of edges in P with $(e_i = (v_i, v_i'), e_j = (v_j, v_j'))$. ∴ we have a YES instance of DEADLOCK, so yes maps to yes. Consider a YES instance of DEADLOCK, we have $v = (v_1, v_2 \dots v_k)$ in deadlock such that no two literals are negations of each other and the literals of each v_i are consistent or there would be an available transition from v to another state. Thus, assigning the literals labeling each $v_1, v_2 \dots v_k$ to TRUE is a satisfying assignment to the 3-SAT. Thus no maps to no. Therefore, DEADLOCK is NP-complete.

b) To prove that REACHABLE DEADLOCK is PSPACE-hard, we will reduce an arbitrary language L that is PSPACE to REACHABLE DEADLOCK in polynomial time. Since L is in PSPACE, there exists a Turing Machine M that decides L in n^k space. We modify M to prevent its head from moving off the start tape. We do this by adding a special character to the beginning of the tape, which shifts the input right by 1, so that any move onto the special character is followed by a move right onto the first tape character. This prevents the tape head from moving off the left side of the tape. Next, we modify M such that if it is going to reject, we enter an infinite loop instead. Thus, this modification simulates M as it never moves outside n^k+1 tape spots — including the start character — and halts if input $x \in L$ and enters an infinite loop if $x \notin L$. Now we construct an instance of REACHABLE DEADLOCK and define $\Sigma' = \Sigma \cup (\Sigma \times Q)$ as the alphabet of the tableau. Let $x = n^k + 1$ and create directed graphs $G_1, G_2 \dots G_x$ each with $|\Sigma'|$ vertices. $\forall\, y \in \Sigma'$ we assign a unique vertex to each G_i to represent y. We then construct P to consist of all pairs $((v_i, v_i'), (v_{i+1}, v_{i+1}'))$ →

2b) for all $v_i, v_i' \in V_i$ and $v_{i+1}, v_{i+1}' \in V_{i+1}$ where $v_i, v_i' \in (\Sigma \times Q)$ and $v_{i+1}, v_i' \in \Sigma$ such that this represents the tape head moving right. Or, conversely, $v_i', v_{i+1} \in (\Sigma \times Q)$ and $v_i, v_{i+1}' \in \Sigma$ represents the tape head moving left. Note that according to M's transition function, v_i, v_{i+1} in one row becomes v_i', v_{i+1}' in the next row of the tableau. This reduction runs in polynomial time. We now show that this reduction is correct. We set the first row of the tableau to the start tape of M where each row in the tableau represents a configuration of M. Row $i+1$ is the next configuration of M of row i in the tableau. Thus, if M accepts input x then $x \in L$. Note that the tableau is of finite length since M halts and there are no further configurations after accepting. Therefore the accept state v can't change and is a deadlock state. The alternative is for M to loop forever such that we will never reach a deadlock state. ∴ REACHABLE DEADLOCK is PSPACE-hard.

3) Since $g(x)$ may be polynomially long in $|x|$, we cannot compute and store $g(x)$ explicitly before evaluating f as that would use $> O(\log n)$ space. Instead, we have to compute $g(x)$ on demand. Let M_g and M_f be TMs to compute g and f using at most $O(\log n)$. We construct a new TM M_h that computes $h(x) = f(g(x))$ in $O(\log n)$ space as follows. M_h simulates M_f with $g(x)$ as its input. Whenever M_f attempts to read the i-th bit of $g(x)$, M_h pauses the simulation of M_f and simulates M_g on input x to compute only the i-th bit of $g(x)$, without storing the entire output of g. To do this, M_h remembers M_f's state, which requires at most $O(\log n)$ space, as well as M_f's position on its input tape, which can be stored in $O(\log n)$ space. M_h runs M_g on input x but ignores what it would have written on an output tape, except for tracking the position of its head and the bit written at the i-th position. Once M_g produces the ith bit of $g(x)$, M_h resumes the simulation of M_f using the retrieved bit. At any step, M_h runs either M_f or M_g but not both. The space used consists of 1. the space required to store the current state and head position of one of the two machines ($O(\log n)$). 2. The space needed to run either M_f or M_g at any time ($O(\log n)$). Since we are only storing the state and head position of one machine at a time, the space is $O(\log n) + O(\log n) = O(\log n)$. Since M_g is a logspace machine, it can run for at most polynomially many steps before halting, so $g(x)$ is at most polynomially long. ∴ we can always compute a required bit of $g(x)$ on demand without exceeding $O(\log n)$. By alternating between M_f and M_g, we ensure that M_h computes $h(x)$ within the logspace. Thus, $h(x) = f(g(x))$ is computable in $O(\log n)$ space.