Problem 1, CS21 Set 7, Matt Lim

To show that the language

$$L = \{y : g(f^{-1}(y)) = 1\}$$

is in NP $\cap$ coNP, we will show that it is both in NP and coNP. To prove this, we can simply give a certificate and polynomial time verifier for both NP and coNP.

To show L is in NP, the certificate is x such that $g(x) = 1$. The verifier will check that $g(x) = 1$, a process which is polynomial in time since g is a polynomial-time computable predicate.

To show L is in coNP, we must simply show that $co - L$ is in NP. We have that, since $g(x)$ is a bit,

$$co - L = \{y : g(f^{-1}(y)) = 0\}$$

To show $co - L$ is in NP, the certificate is x such that $g(x) = 0$. The verifier will check that $g(x) = 0$, a process which is polynomial in time since g is a polynomial-time computable predicate.

So we have shown that L is in NP and coNP. So L is in NP $\cap$ coNP.

Problem 2, CS21 Set 7, Matt Lim

(a) To show

$$\text{DEADLOCK} = \{(G_1, G_2, \ldots, G_n, P) : \text{directed graphs } G_1, G_2, \ldots, G_n \text{ and the list } P \text{ specify a system with a deadlock state.}\}$$

is NP-complete, we must show that it is in NP and that all NP-problems are polynomial time reducible to it. To prove the first part, we can simply give a certificate and a polynomial time verifier. The certificate is a deadlock state $v \in V$. The verifier will verify that v is a deadlock state by checking if there is a $w \in V$ such that $(v, w) \in T$. If there is no w such that $(v, w) \in T$, then v is a deadlock state. This process is clearly polynomial since the number of states $w \in V$ is polynomially large with respect to the input.

To prove the second part, we show that 3-SAT is polynomial time reducible to DEADLOCK. The following details the reduction.

Our reduction function f will map a 3-CNF formula φ to $A = (G_1, G_2, \ldots, G_n, P)$, where G_i is a directed graph and P is a list of pairs of edges, as described in the description of the problem. We will map φ to A in the following way. For each clause, we will make a process/directed graph. The process/directed graph will be in the form of a triangle, with each vertex of the triangle representing a literal in the clause, and with directed edges such that each vertex/state can go to the other to vertices/states. So, given a clause $(x_1 \vee x_2 \vee x_3)$, we would generate a graph G_1 with vertices v_{11} (representing x_1), v_{12} (representing x_2), and v_{13} (representing x_3), where each vertex/state would be able to travel to the other two vertices/state. We will then generate list P in the following way. For each literal x_i, if $\neg x_i$ appears in a separate clause, then we will add the pair $((x_i, a), (\neg x_i, b))$, where a is a literal in the same clause as x_i and b is a literal in the same clause as $\neg x_i$. If there exist multiple clauses with x_i or multiple clauses with $\neg x_i$, then we will add in a corresponding pair $((x_i, a), (\neg x_i, b))$ to P for each additional x_i and $\neg x_i$ pair.

Now we will show that this reduction function f maps yes to yes and no to no. First we will show it maps yes to yes. So, given that φ has a satisfying assignment, we want to show that A has a deadlock state. Since φ is satisfiable, there is at least one true literal in each clause. So the state will be made up of one vertex from each graph that corresponds to a true literal in the graph's corresponding clause. First we will consider the case in which φ has all x_is and no $\neg x_i$s. Then, our P will be empty, so there will be no available transitions, and A will have a deadlock state. Next we will consider the case in which φ does have x_is and $\neg x_i$s, but each x_i and $\neg x_i$ pair is in its own clause. Then our P is empty again, so A will have a deadlock state. Finally, we will consider the case in which φ does have x_i and $\neg x_i$ pairs in different clauses/processes. In this case, we also have that A has a deadlock state. This is because, since φ is a satisfiable assignment, only one of x_i and $\neg x_i$ can be true. So WLOG, let x_i be true. Then, given how we defined f, in one process we have the transition (x_i, a) (where a is a literal/vertex in the same clause/process as x_i) and in another process we have the transition $(\neg x_i, b)$ (where b is a literal/vertex in the same clause/process as $\neg x_i$). We also have that the pair of transitions $((x_i, a), (\neg x_i, b))$ is in P. But now we have that there is no w such that $(c, w) \in T$, where c is a literal/vertex in the same clause/process as $\neg x_i$. This is because for $(c, w) \in T$ to be true, $((x_i, a), (c, w))$ must be in P. But we have that only pairs of the form $((x_i, a), (\neg x_i, b))$ are in P. This logic can be extended to all the other clauses. In other words, the only transitions that can happen are those that involve transitioning from x_i to something and $\neg x_i$ to something. But the state $v \in V$ (as defined at the top of this section) is made up of only one of x_i and $\neg x_i$, not both of them. So there can be no transitions. Thus we have that A has a deadlock state.

Now we will show that f maps no to no. To prove this, it suffices to show that if A is in DEADLOCK, then φ is satisfiable. We have that our A has a deadlock state. So we can simply pick all the literals that correspond to the vertices specified by the deadlock state. So we have that at least one literal in each clause is true, since the state is made up of one vertex from each graph. We also have that x_i and $\neg x_i$ cannot both be in the state, since if they were, then the state (due to how we defined f) would not be a deadlock state. So we have that we can construct a satisfiable assignment for φ and thus we have that no maps to no.

Now we must only prove that our reduction function f is polynomial time computable. For each clause f generates a process/directed graph, a process which is clearly polynomial. And for each pair $x_i, \neg x_i$ in different clauses, f adds a pair to P, a process which is also clearly polynomial (bounded by the number of clauses to some power k). So we have that our reduction function is polynomial.

Finally, we can conclude that DEADLOCK is NP-complete.

(b) To prove that REACHABLE DEADLOCK is PSPACE-hard, we will reduce from a general L in PSPACE to REACHABLE DEADLOCK. We will do this by representing L and its corresonding TM M as a tableau. We will have the rows of our tableau represent the current configuration of our TM M that recognizes L. These rows will be of polynomial length, since L is polynomial space. Our tableau language will be the union of our tape language, as well as all the symbols of the form (a, q), where a is a tape symbol and q is the state (so the head is at the tape symbol a in this case). The first row of our tableau will be the initial state of M. We will then define our transition functions as follows. Basically, the head can either move left, right, or stay put. If our initial configuration looks like $a, (b, q), c$, then our left transition functions will be of the form $(a, q'), b', c$, where b' is in the tape alphabet (possibly different than b) and q' represents a state (possibly different than q). The right transition functions will be of the form $a, b', (c, q')$, where b' is in the tape alphabet (possibly different than b) and q' represents a state (possibly different than q). The stay put transition functions will be of the form $a, (b', q'), c$, where b' is in the tape alphabet (possibly different than b) and q' represents a state (possibly different than q). For all these transitions, symbols that are not shown are not affected. We will also have that the accept state q_{accept} does not transition to anything, and that q_{reject} goes to q'_{reject}, which then goes back to q_{reject}, creating an infinite loop (note that we will not be adding an infinite number of rows; we will just loop back and forth between two rows). Now we will take our tableau and reduce it to REACHABLE DEADLOCK. So we will let each column represent a graph G_i. Then we will let P be the list of transition functions, modified to fit what P is supposed to be (a list of pairs of edges). So for a given transition, we could map that to one pair $((e_1, e'_1)) \in P$ that is viable given the overall state. We will have each cell represent the state of a graph (the graph being the column label), so each row represents the overall state of all the graphs.

Now we will show that yes maps to yes and no maps to no. First we will show that yes maps to yes. So, given that a string x is in L, that means our TM M accepts. And since we said that once q_{accept} is reached there are no more transitions, this means that we cannot transition from one overall state of the graphs to another. So deadlock is reached once we reach the accept state.

Now we will show that no maps to no. So given that a string x is not in L, M rejects. This means q_{reject} is reached, which means we enter an infinite loop. So there will always be transitions for the overall state of the graphs to take, and deadlock is never reached.

Now we must only prove that our reduction takes polynomial time and space. Our reduction function maps, for each row, at most n^k cells, where n is the input length to M, to at most n^k states, each of which is some constant number of vertices. And there are, in terms of the input length n, a constant number of rows. So the reduction function is polynomial in both space and time.

Finally we can conclude that REACHABLE DEADLOCK is PSPACE-hard.

Problem 3, CS21 Set 7, Matt Lim

We have that $f(w)$ and $g(w)$ are both computable in space $O(\log n)$. So to show that $h(w) = f(g(w))$ is computable in $O(\log n)$, we will do the following. The first tape of our TM M will just be the input w. The third tape of M will be the output $h(w)$. The first part of the second tape will contain the work needed to compute $g(w)$, which takes up $O(\log n)$ space. Now we must only show that the work needed to compute $f(g(w))$ also only takes up $O(\log n)$ space. To make this happen, we will compute $f(g(w))$ bit by bit. That is, when computing $f(g(w))$, we will take one bit of $g(w)$, work with that, and then output the result onto the third tape. So, back to the second tape. As we said before, the first part of the second tape contains the work needed to compute $g(w)$. This is $O(\log n)$ space. The next part will contain the relevant bit of $g(w)$ that we are working with. We can have the functionality of telling which bit of $g(w)$ that we need built into M, and we can retrieve the relevant bit using the first part of the second tape (the work needed to compute $g(w)$. This is obviously in $O(\log n)$ space. Finally, we will have the work need to compupte $f(g(w))$, where $g(w)$ is the bit written on the second part of the tape. We will now argue that $g(w)$ is in polynomial space. By considering that M has a finite number of states and a finite sized alphabet, we can see that the output $g(w)$ has space of order $O(A^{k\log n})$, where A is the size of the alphabet, k some constant, and n the length of w. So we have that the space of the output $g(w)$ is polynomial. Then we have that, since f is computable in $O(\log n)$ space, then $f(g(w))$ is computable in $O(\log n)$ space (since the size of $g(w)$ is polynomially large). So we have that our second tape is taken up by 3 things that are $O(\log n)$ space, which means that it is $O(\log n)$ space for each step (for each bit retrieved and used). And after each step is done, we can just wipe the last two parts (the bit of $g(w)$ and the work need to compute $f(g(w))$) and repeat the process until $f(g(w))$ is calculated. Thus we have that $h(w) = f(g(w))$ is computable in $O(\log n)$ space.