Problem 1, CS21 Final, Matt Lim

(a) We will prove that L is in PSPACE. To do this, we will consider the algorithm that decides this problem. For the odd numbered moves (player 1), we can do the following. For both moves S_i and T_i, we can recursively solve the game for the situation where player 2 gets to choose from S_{i+1} or T_{i+1} (with the chosen subset S_i or T_i having been deleted from G, along with their incident edges). If at least one of the moves of player 1 returns a "win" for player 1, return "yes". Else, return "no". For the even numbered moves (player 2), we can do the following. For both S_j and T_j, we can recursively solve the game for the situation where player 1 gets to choose from S_{j+1} or T_{j+1} (with the chosen subset S_j or T_j having been deleted from G, along with their incident edges). If at least one of the moves from player 2 doesn't return a "win" for player 1, return "no". Else, return "yes". The base case is just determining whether or not a graph contains a k-clique. This can clearly be done in polynomial space by simply looking at all combinations of k or more vertices one at a time and determining whether any are cliques. The recursion depth is also polynomial, as is the bits of state at each level (we actually lose a pair of subsets at each level of recursion). So we have that L is in PSPACE.

(b) We will prove that L is PSPACE-hard. To do this we will reduce QSAT to L. Our reduction function f will be as follows. For each clause in φ, we will generate a triple of nodes. If a clause has less than two nodes, then our triple will simply contain duplicates of the literals in that clause. We will then put all possible edges between different triples. We will let the subset of vertices x_i be S_i, and the subset of vertices $\neg x_i$ be T_i. Then we will let $k = m$, where m is the number of clauses in the instance of QSAT we reduce from. Observing this reduction function, it is clearly polynomial. Let's go through each step to make sure. For each clause, we generate a triple of nodes, a polynomial process. Then we put all possible edges between the different triples, also a polynomial process. Finally, we map each set of vertices x_i to a S_i and each set of vertices $\neg x_i$ to a T_i. All these processes are polynomial, so we have that our reduction function f is polynomial.

Now we will show that this reduction function f maps yes to yes and no to no. First we will show yes maps to yes. So, given that φ is in QSAT, we want to show that the $(G, k, (S_1, T_1), \ldots, (S_n, T_n))$ it maps to is an instance of L. So, assume that φ is a yes instance of QSAT. This means $\exists x_i \forall x_2 \exists x_3 \forall x_4 \exists x_5 \ldots \forall x_n$ such that φ is satisfied. So basically, "player 1" can satisfy φ no matter what "player 2" does. Now we will observe how this translates to our graph. For each literal value that we pick, we delete a corresponding subset of vertices. Consider what happens when we pick the opposite set of vertices; that is, if x_i is true in φ, we will pick the subset of $\neg x_i$s to be deleted. Since there must be at least one true literal in each clause (since φ is satisfiable), at the end of the game, there will be at least m vertices, one from each triple. And each of these m vertices will be connected to each other, since we put all possible edges between different triples. So the graph will have a clique of size m no matter what. And since we let $k = m$, we have that $(G, k, (S_1, T_1), \ldots, (S_n, T_n))$ is a yes instance of L. Thus yes maps to yes.

Now we will show that this reduction function f maps no to no. To do this, it suffices to show that if $(G, k, (S_1, T_1), \ldots, (S_n, T_n))$ is in L, then φ is in QSAT. So, assume we have that $(G, k, (S_1, T_1), \ldots, (S_n, T_n))$ is in L. This means that player 1 can win no matter what player 2 does. So player 1 can make it so that at the end of the game, there is always a k-clique. But there can only be a k-clique if at least one vertex from every triple of vertices remains, because the triples of vertices aren't connected within themselves. And we have that, of the remaining vertices, there cannot be any contradictory vertices (because each turn we either delete all x_i's or all $\neg x_i$'s). This means that the vertices at the end of the game represent a satisfying assignment for φ. So in total, we have that player 1 can always make a k clique appear at the end of the game, which in turn means that player 1 can always satisfy φ. Thus no maps to no.

We proved above that our reduction function f is polynomial. So finally, we can conclude that L is PSPACE-hard.

Problem 2, CS21 Final, Matt Lim

We will prove that L is context-free but not regular. First we will prove that L is not regular. To do this, we can use the fact that regular languages are closed under complement. This means that if the complement of L is not regular, then L itself is not regular. We have that the complement of L, $\overline{L}$, is all strings with an equal number of a's and b's (and any number of c's). We will prove that $\overline{L}$ is not regular by using the pumping lemma.

Assume that $\overline{L}$ is regular. Then there exists a pumping length p. We will choose the string $w = a^p b^p c \in \overline{L}$. We can write this string as $w = xyz$, with $|y| > 0$ and $|xy| \leq p$. Since $|xy| \leq p$, then y must be contained in the first p a's (i.e. y can only be in the a's). So, choose any y in the first p a's. Pumping on y gives us an unequal number of a's and b's. So pumping on y gives us a string that is not in the language. But this is a contradiction. Thus we have that $\overline{L}$ is not regular, and thus that L is not regular.

Now we will prove that L is context free using the following statement: L is recognized by a NPDA implies L is described by a CFG. We will construct an NPDA that recognizes L, thus proving L is context free. Look at the NPDA constructed on the following page. Let $k \neq a, b, c$ and let m be any symbol. As we can see, L is recognized by our NPDA. Thus, L is context free.

Problem 3, CS21 Final, Matt Lim

To show that L is neither R.E. nor co-R.E., we will show that EQ_{TM}, which is neither R.E. nor co-R.E. (as proved in the book and in lecture) is reducible to L. The reducing function f works as follows. On an input $\langle M_1, M_2 \rangle$, we will map this to output $\langle M_2, M_1, M_2 \rangle$, where M_1 and M_2 are the same as they are in the input. So, if $\langle M_1, M_2 \rangle$ is a yes instance of EQ_{TM}, then $L(M_1) = L(M_2) \implies L(M_2) \subseteq L(M_1) \subseteq L(M_2)$, which means that $\langle M_2, M_1, M_2 \rangle$ is a yes instance of L. So yes maps to yes. If $\langle M_1, M_2 \rangle$ is a no instance of EQ_{TM}, then we have three possible cases. The first is the case in which $L(M_1)$ contains a string (or strings) not in $L(M_2)$. Then we have that $L(M_1) \nsubseteq L(M_2)$, and thus that no maps to no. The second is the case in which $L(M_2)$ contains a string (or strings) not in $L(M_1)$. Then we have that $L(M_2) \nsubseteq L(M_1)$, and thus that no maps to no. The third is the case in which $L(M_1)$ contains a string (or strings) not in $L(M_2)$ and $L(M_2)$ contains a string (or strings) not in $L(M_1)$. So we have that neither of them are subsets of each other (or equal to each other) and thus that no maps to no. So we have reduced EQ_{TM} to L and thus shown that L is neither R.E. nor co-R.E..

Problem 4, CS21 Final, Matt Lim

(a) Assume that the language L is regular. Now we will show that the language

$$L_{-y} = \{xz : x \in \Sigma^* \text{ and } z \in \Sigma^* \text{ and } xyz \in L\}$$

which consists of all strings in L with the string y deleted from them is also regular. To do this, we will use the theorem that a language L is recognized by a FA if and only if L is described by a regular language. So, from this theorem, we have that L is recognized by some FA. Now we must only shows that L_{-y} is also recognized by some FA (or some NFA, since FAs and NFAs are equivalent). To do this, we will consider the FA that recognizes L. Let us call it A. We have that, since A recognizes an arbitary $xyz \in L$, there is some part of A, composed of states and transitions, that reads through $x \in \Sigma^*$, some part of A that reads through $y \in \Sigma^*$, and some part of A that reads through $z \in \Sigma^*$. Now, consider the state that A is in when it reaches the last symbol of x. This state must have some transition that reads the first symbol of y. Now, consider what happens if we delete this transition, and add a transition that reads the first symbol of z. We effectively now have a FA that reads to the last symbol of x, skips the y string entirely, and transitions from the last symbol of x to the first symbol of z. In other words, we have a FA that recognizes L_{-y}. So we can conclude that L_{-y} is regular.

(b) Assume that L is R.E.. Now we will show that L_{-y} is also R.E.. We have that a language is R.E. if it is recognized by some TM. We will let M_L be the machine that recognizes L. Now we must design some machine M that recognizes L_{-y}. We will fix some enumeration $y_1, y_2, \ldots$ of Σ^* in lexicographic order. Our machine M for L_{-y} does the following for $i = 1, 2, 3, \ldots$: simulates machine M_L on the inputs $xy_1z, xy_2z, \ldots, xy_iz$ for i steps, for each way we can split xz into $x \in \Sigma^*$ and $z \in \Sigma^*$. If during any of the simulations M_L accepts some xy_jz, then M halts and accepts.

Suppose xz is in L_{-y}. Then there is some xyz (for some splitting x and z of xz) which is accepted by M_L after n_L steps. If y appears in the lexicographic order at position m, then our machine M will accept when $i = max(m, n_L)$. Thus every string $xz \in L_{-y}$ is accepted by M.

Conversely, if M accepts some string xz, then it can only have done so because for some value i, there was some string y_j with $j \leq i$ for which M_L accepted xy_jz for some splitting x and z of xz. Therefore there is a string xy_jz in L for some splitting x and z of xz, which implies $xz \in L_{-y}$ as required.

Problem 5, CS21 Final, Matt Lim (a, b, d, e)

(a) We will show that the language L described in part (a) is NP-complete. To do this, 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 just an independent set of G with size at least k. The verifier will iterate through the vertices in the set and check that no pair of vertices makes up an edge in G. It will also iterate through all vertices of G and make sure that no vertex has more than 4 edges touching it. Both operations are clearly polynomial.

To prove the second part, we show that INDEPENDENT SET (IS) is polynomial time reducible to L. The following details the reduction.

Our reduction function f will map (G, k) (a graph and an integer) to (G', k') (another graph and an integer. It will do this in the following way. We will let $k' = k$, so we will just copy over k. To make G', we will do the following. We will iterate through all the vertices of graph G, and for each vertex that has more than 4 edges, we will delete edges from it until it only has 4 edges. This "trimmed" graph will be our G'.

Now we will show that this reduction function f maps yes to yes and no to no. First we will show that yes maps to yes. So, assume that (G, k) is in IS. That means it has an independent set of size at least k. Then we have that our G' also has an independent set of size at least k, since deleting edges from G cannot possibly decrease the size of its independent set (by the definition of an independent set). And since $k' = k$, and since we trimmed the number of edges so that G' has no vertices with more than 4 edges that touch it, we have that (G', k') is in L.

Now we will show that this reduction function f maps no to no. To do this, it suffices to show that if (G', k') is in L, then (G, k) is in IS. So, assume (G', k') is in L. Then we have that G' has an independent set of size at least k'. Now consider its corresponding G. G' is basically the same as G, except with some edges trimmed. So let us consider what happens when we add those edges back to form G. At least k' vertices in an independent set of G' are not affected at all, since those same vertices in G had no edges between them. And since $k' = k$, we have that G has an independent set of size at least k. So (G, k) is in IS and no maps to no.

Now we must only prove that our reduction function f is polynomial time computable. Our function is clearly polynomial. We simply iterate through all the vertices (polynomial), delete a polynomially large number of edges from each vertex, then map that trimmed graph and an integer k to a new graph and new integer k'. So we have that our reduction function is polynomial.

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

(b) We will prove that the language described in part (b) is in P by giving a polynomial algorithm for it. Consider the case when k is over 101. Then we have that (G, k) cannot be in L, because since G must have a maximum degree of 100, the maximum clique it can have is a 101-clique. Now consider the case when k is 101 and below. Then we can treat k as a fixed number, and solve for the fixed numbers 1-101. For each fixed number m (and letting y be the number of vertices in G) we can try all the yCm ways that m vertices can be chosen in G. This is $O(y^m)$ time. Then, for each way that m vertices can be chosen, we will see if they form a clique. This is clearly polynomial because there are a polynomially large number of edges in G. Then we will record our result for each m. So since for each m the process is polynomial, then the process as a whole is also polynomial. Then at the end, we will iterate through all the results until we reach the kth one. If that value is yes, we accept. If it is no, we reject.

(d) We will show that the language described in part (d) is in P by reducing it to 2-SAT, which is in P.

We will call the language in part (d) L. So, take an arbitrary ϕ in L. Our reduction function f will map this to a 2-CNF formula φ. It will do this by trying all possible ways to put 750 of the first 1000 clauses along with the rest of them. So, it will try all 1000 choose 750 ways to choose 750 of the first 1000 clauses (then append the rest of the clauses to the chosen 750) and see if they are in 2-SAT (we proved that deciding 2-SAT is polynomial time in class). The first time that the chosen set of clauses is in 2-SAT, f will map those set of clauses to be φ. If none of the chosen set of clauses is in 2-SAT, f will just copy over ϕ to φ.

Now we must show that f maps yes to yes and no to no. First we will show that it maps yes to yes. So, assume ϕ is a yes instance of L. Then we have that there is some set of 750 clauses from the first 1000 clauses that is completely satisfiable by the same assignment that satisfies all the other clauses. By our reduction function, this set of satisfiable clauses (the 750 plus all the other ones) is φ. So φ is satisfiable and thus is in 2-SAT. Thus we have that yes maps to yes.

Now we will show that this reduction function f maps no to no. So, assume ϕ is a no instance of L. That means that there does not exist an assignment that satisfies all the clauses. And since in this case, our reduction function f just copies ϕ over to φ, then there is no satisfying assignment for φ. So no maps to no.

Now we must only prove that our reduction function f is polynomial time computable. The first process f goes through is trying all 1000 choose 750 ways of picking 750 from the first 1000 clauses. This is constant time in regards to the input length. Then, it simply maps a set of clauses from ϕ to φ, a clearly polynomial process. Thus we have that the reduction function is polynomial.

Finally, we can conclude that L is in P.

(e) We will show that the language L described in part (e) is NP-complete. To do this, 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. If ϕ is a 2-CNF formula, the certificate is simply an assignment that satisfies all the first 1000 clauses of ϕ and at least 3/4 of the other clauses. The verifier will check that this assignment indeed does satisfy the desired clauses. It can do this by simply iterating through the first 1000 clauses and seeing if they are all satisfied, then iterating through the rest, keeping a counter of how many total other clauses there are and a counter of how many of those other clauses are satisfied. It will then check that at least 3/4 of those other clauses were satisfied by the assignment. The verifier will also make sure that the 2-CNF formula has at least 1000 clauses, and that it has at most 2 literals per clause. To prove the second part, we show that 3-SAT is polynomial time reducible to L. The following details the reduction.

Our reduction function f will map a 3-CNF formula φ to a 2-CNF formula ϕ. It will do this in the following way. First, we will make some arbitrary clause $(q \vee q)$ (where q is unique with respect to the rest of the clauses not in the first 1000) repeat 1000 times. Next, for each clause in φ, we will construct 12 new clauses with 2 literals. Let us consider a single clause $(x, y, z) \in \varphi$. We will map this clause to the 10 new 2-literal clauses shown below:

$$(x \vee x), (y \vee y), (z \vee z), (w \vee w),$$

$$(\neg x \vee \neg y), (\neg y \vee \neg z), (\neg z \vee \neg x),$$

$$(x \vee \neg w), (y \vee \neg w), (z \vee \neg w)$$

$$(a \vee a), (a \vee a)$$

Consider the truth table for these 12 clauses, as shown below, letting 1 represent true, 0 represent false, and max be the max number of clauses that are simultaneously satisfiable (depending on what w is assigned and on what a is assigned):

x	y	z	max
1	1	1	9
1	0	0	9
0	1	0	9
0	0	1	9
1	1	0	9
0	1	1	9
1	0	1	9
0	0	0	8

Note that the maximum number of clauses that can be satisfied here is 9. Also note that this happens exactly when $(x \vee y \vee z)$ is true. So, this is our reduction function.

Now we will show that this reduction function f maps yes to yes and no to no. First we will show yes maps to yes. So, given that φ has a satisfying assignment, we want to show that ϕ is in L. So, assume φ has a satisfying assignment. The first 1000 literals in ϕ are all trivially satisfiable by setting q to be true. And they are satisfiable no matter what the other clauses are, since q

only appears in the first 1000 clauses. Now, for the other clauses. We have that for each clause in φ, $(x \vee y \vee z)$ is true. This means that for each of the 12 clauses we generate to put in ϕ from a single clause in φ, 9/12 of them can be made true. And this is true for every set of 12 clauses generated, which makes up all the other clauses in ϕ. So we have that 9/12 of all the other clauses are satisfiable, which means that at least 3/4 of the other clauses are satisfiable. So we have that there is an assignment that satisfies all the first 1000 clauses (trivially) and at least 3/4 of the other clauses.

Now we will show that this reduction function f maps no to no. So, consider the other clauses (the ones after the first 1000). We have that φ is not satisfiable. This means that $(x \vee y \vee z)$ cannot be made true for every clause in φ. This means that, for at least one set of 12 clauses in ϕ, less than 9 clauses can be made true. But since the maximum number of clauses that can be made true for every set of 12 is only 9, this means that less than 9/12 of all the other clauses can be made true. So we have that ϕ is not in L and thus that no maps to no.

Now we must only prove that our reduction function f is polynomial time computable. Our function is clearly polynomial, because for each clause in φ, it simply generates 12 clauses to put in ϕ. So we have that our reduction function is polynomial.

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