Problem 1, CS38 Midterm, Matt Lim

(a) See attached sheet.

(b) We will assume the graph has no self-loops. We will also assume that there are no repeats of edges for convenience, since adding them or removing them does not matter for detecting triangles. Here is pseudo-code for our algorithm. Let A be the adjacency matrix of the graph.

1. We begin by squaring A to get the matrix A^2. This is $O(n^{\log_2 7})$, which is less than $O(n^3)$.
2. We then iterate down the diagonal of A^2. For each column/row i that has a value v greater than or equal to 2 on the diagonal, we will label the corresponding row and column on the matrix A with v. We will label every other column and row with 0. This label corresponds to how many edges are going into a given node (so if a row is labeled with 5, there are 5 edges going into it). And given our initial assumptions, it also represents how many vertices a given node is connected to. This takes time $O(n)$. Note that from here on, we will be working with the matrix A.
3. Then, for each row with a value greater than or equal to two, we will iterate through its columns. First, we will reset its value to zero. Then, for each of its columns (with an index not equal to the current row index) with a nonzero value in it, we will add one to the row's value. After we have finished reassigning the row values, we will copy those values over to the corresponding columns. We will then sort the rows again in ascending order. This is $O(n^2 \log n)$.
4. We will iterate through the rows in ascending order. For each row R with a value greater than or equal to 1, we will iterate through its columns. Let i be the index of R. If we encounter a nonzero value in a column that a) is labeled with a nonzero value, b) is not marked by X (see the end of this item), and c) whose index is not i, we will store its index in a list L. Then, having iterated through each column of R, we will do the following. For every pair of indices j_1 and j_2 in L, we will check the (j_1, j_2) position of the matrix. If there is a one at that position, then our graph has a triangle. If we do this for every pair of indices and we find no triangle, then we will mark column i with an X.
   - We claim that this step runs in $O(n^2)$. To see why, consider the following argument. When iterating through any column, we will never generate a list with more than 2 items. This is because we go through in ascending order and because we only add items from columns that are labeled with a nonzero value. It is also because we X out columns once we discover that their corresponding node is not in a triangle.
     Basically, we must always have at least one node of nonzero value that is attached to at most two nodes of nonzero values that are not crossed out. That is, we must always have at least one node with a value of at most two and at least one. The only other cases are when there is no such node, in which case we are done iterating through the rows, or when the graph is infinite, which we do not have.
     Since we never generate a list with more than 2 items, then going through each row only takes $O(n)$. And since there are n rows, we get time $O(n^2)$ for this step.

If we follow through the steps, we can see that this algorithm runs in the desired time of $O(n^\alpha)$, for some $\alpha < 3$.

So we have given the algorithm and given time bounds. Now we will explain why this algorithm works. The first three steps label each row and column with a number. Let us look at row R that represents node v with number l. The number l corresponds to how many neighbors v has that are connected to two or more nodes. This number is important because if a node does not have at least two neighbors that are connected to two or more nodes, then it cannot be in a triangle. We will call nodes that have at least two neighbors that themselves are connected to two or more nodes "number" nodes. After labeling the rows and columns, we then sort the rows in ascending order. This is so that we go through the rows with the lowest label first. It is in step 4 that we iterate through the rows. For each number node/row, we generate a list of neighbor nodes who are also number nodes. Note that we exclude crossed out columns/nodes from this list. We only add number nodes because they are the only nodes that can be in a triangle. Then, for each pair in the list, we check to see if their is an edge between them. If so, we have found a triangle. Overall, what this step does is check every possible relevant triple of nodes. Thus, it is clear that our algorithm

computes the right answer. Note that after we determine that a node does not contain a triangle, we mark its corresponding column with an X. This is so that when we are looking at neighbors of this node, we don't consider it anymore or put it in the list, since we already know it is not in a triangle.

Problem 2, CS38 Midterm, Matt Lim

(a) We will assume that every element $e \in E$ is mentioned in some subset of I. This means that for every element $e \in E$, the set $\{e\}$ is in I (by the second axiom of matroids). Now we will show that the set system on $E' = E \backslash \{e\}$ defined by

$$I' = \{A : (A \cup \{e\}) \in I\}$$

is a matroid.

First we will show that axiom one holds. For $\varnothing \in I'$ to be true, then $\varnothing \cup \{e\} = \{e\} \in I$ must be true. And this indeed is true by the assumption we made. So axiom one holds.

Now we will show that the second axiom holds. Let us assume that $A \in I'$. Now we must show that every subset $B \subseteq A$ is also in I'. So, since $A \in I'$, we have that $A' = A \cup \{e\} \in I$. This means that every subset $B' \subseteq A'$ is also in I, since I is a matroid. Then it follows that every subset of A is in I. It also follows that every subset of A unioned with $\{e\}$ is in I. This means that every subset of A is in I'. So the second axiom holds.

Finally, we will show that the third axiom holds. So, let A and B be subsets in I' with $|B| > |A|$. We must show that there exists $x \in B \backslash A$ such that $A \cup \{x\}$ is in I'. Let us consider $A' = A \cup \{e\} \in I$ and $B' = B \cup \{e\} \in I$. Then we have that there exists $x \in B' \backslash A'$ such that $A' \cup \{x\}$ is in I, where $x \neq e$. Then it follows that we can union this x with our original A to make $A \cup \{x\}$. Then we can say that $A \cup \{x\}$ is in I', since $A \cup \{x\} = (A' \cup \{x\}) \backslash \{e\}$. Thus all three axioms hold and we have that I' is a matroid.

(b) Note that for this problem, we will assume that F does not contain a cycle. We will first argue that the subsets of G's edges that, together with F, form spanning trees, constitute the bases of a matroid I. So, let these subsets be the bases of I (they have the same cardinality because all spanning trees have $|V| - 1$ edges and they are clearly independent sets), and let I contain all subsets of them. We want to show that this I is a matroid with the desired bases.

Let F have n edges, $f_1, f_2, \ldots, f_n$. Now consider the matroid I_n, whose bases are the spanning trees of G that contain F. We have that all subsets of these bases are acyclic, since they are subsets of spanning trees. Clearly I_n is indeed a matroid, since it is a graphic matroid. Now, consider the matroid $I_{n-1} = \{A : (A \cup \{f_1\}) \in I_n\}$ with universe $E' = E \backslash \{f_1\}$. We have from part (a) that this a matroid. Then we can continue on in this fashion (forming $I_{n-2} = \{A : (A \cup \{f_2\}) \in I_{n-1}\}$, etc), continuing to remove edges of F, until we get the matroid I, whose bases are the subsets of G's edges that, together with F, form spanning trees. The universe of this matroid is clearly $E_I = E \backslash F$. Thus we can conclude by part (a) that I is indeed a matroid.

Now we will use I to find the desired minimum cost tree.

Here is a way to formulate this minimum cost spanning tree problem as the problem of finding a maximum weight independent set in I. We will give nonnegative integer weights for each element of the universe $E \backslash F$, which will correspond to the edge weights of G. That is, each element in the matroid will represent an edge in the graph, and will be given a positive integer weight. However, these edge weights will not be the same as the edge weights of the graph. Instead, we will "flip" the weights. That is, we will take the maximum edge weight from the graph, add one to it, and subtract the weight of each edge to find the new weight of each edge. So, if the max edge weight is w_{max}, and we have an edge with weight w in the graph, the corresponding weight in the matroid is $w_{matroidedge} = w_{max} + 1 - w$. We can rewrite this as $w_{matroidedge} = w_0 - w$, where $w_0 = w_{max} + 1$. Then we have that all weights are positive, and that finding the maximum weight independent set in this matroid will give the minimum cost set of edges that, together with F, forms a spanning tree. To see a short proof of this, consider the following. Each maximal independent subset $A \in I$ corresponds to a set of $|V| - 1 - |F|$ edges. Then we have the following (where $w'(e)$ represents the weight of the edge e in the matroid, $w(e)$ represents the weight of the edge e in the graph, $w'(A)$ represents the sum of the matroid edge weights, and $w(A)$ represents the sum of the graph edge weights):

$$w'(A) = \sum_{e \in A} w'(e)$$

$$= \sum_{e \in A} (w_0 - w(e))$$

$$= (|V| - 1 - |F|)w_0 - \sum_{e \in A} w(e)$$

$$= (|V| - 1 - |F|)w_0 - w(A)$$

Clearly, maximizing $w'(A)$ must minimize $w(A)$.

Now we will prove time bounds and correctness of the greedy algorithm. Assigning weights to all the edges is $O(m)$. Then, we proved in set number 2 problem number 2b that the greedy algorithm to find the maximum weight independent set in a matroid is $O(m \log m)$, where m is the number of edges in G. In that same problem we also proved correctness. Since we have that our greedy algorithm is correct, and given the way we defined our edge weights in I, it follows that finding a maximal independent subset A of I is the same as finding the minimum weight subset of G's edges that form a spanning tree with F. Then we can just add F's edges to our found subset and obtain the minimum cost spanning tree containing F. Thus our proof is complete.

Problem 3, CS38 Midterm, Matt Lim

(a) Here is our algorithm.

1. First we will sort each list. This is $O(n \log n)$.
2. We will then make n new lists. Let $A = a_1, \ldots, a_n$, $B = b_1, \ldots, b_n$, and $C = c_1, \ldots, c_n$ be the newly sorted lists. Then we will have $L_1 = a_1 + b_1, a_1 + b_2, \ldots, a_1 + b_n$, $L_2 = a_2 + b_1, a_2 + b_2, \ldots, a_2 + b_n$, etc. etc. up until $L_n = a_n + b_1, a_n + b_2, \ldots, a_n + b_n$. This is $O(n^2)$.
3. Note that all the lists we made in the previous step are sorted. Thus we can do the following. We will merge all our lists into one big sorted list. We will call this list L. We can see that L will have order n^2 elements. Thus this step is $O(n^2)$.
4. Then, we will iterate through list C. For each element $c_i \in C$, we will binary search list L for $-c_i$. If this search succeeds for any c_i, then there exist i, j, k such that $a_i + b_j + c_k = 0$. This is time $O(n \log n^2) = O(n \log n)$.

Observe that the dominating time bound above is $O(n^2)$. Thus we have that our algorithm runs in time $O(n^2)$. So we have given the algorithm and shown the time bound for it. Now we will show correctness. We have that our list L contains every possible sum of one element from A and one element from B. In other words, L contains every possible $a_i + b_j$. Therefore, we need only to know if any element from L plus any element from C sums to 0. That is, each must check if there exists i, j such that $l_i + c_j = 0$, $l_i \in L$. Our algorithm does exactly this, checking if if there exists an $l_i = -c_j$ for every $c_j \in C$. Thus it follows that our algorithm works correctly.

(b) Note that for this problem we will assume that $M > n$. Now here is our algorithm.

1. Let $A = a_1, \ldots, a_n$, $B = b_1, \ldots, b_n$, and $C = c_1, \ldots, c_n$ be our lists. We will begin by forming two polynomials:
$$P_1(X) = \sum_{i \in A} X^{i+M}$$
$$P_2(X) = \sum_{i \in B} X^{i+M}$$
Shifting the degree ensures we have no negative degrees. Forming these polynomials is time $O(M)$.
2. We will then perform fast polynomial multiplication, and form a new polynomial:
$$P(X) = P_1(X) \cdot P_2(X)$$
This is $O(M \log M)$ using fast polynomial multiplication as seen in lecture, since the largest degree possible is $2M$ and the smallest possible degree is 0. This is because all of the integers lie in the range $[-M, M]$ and because of how we shifted the degrees in the step above.
3. We will then iterate through $P(X)$ and form a list L of all the degrees/powers, with each one shifted down by $2M$. This is because we originally shifted all the powers up by M. We will assume that $P(X)$ has its terms sorted in order of ascending degree (e.g. $x + x^2 + x^4 + x^7 \ldots$). Then we have that L is sorted in ascending order. Making this list is $O(M)$, since our polynomial has maximum degree $2M$ and minimum degree 0, and we are assuming that $M > n$.
4. Then, we will iterate through list C. For each element $c_i \in C$, we will binary search list L for $-c_i$. If this search succeeds for any c_i, then there exist i, j, k such that $a_i + b_j + c_k = 0$. This is time $O(M \log M)$, since we are assuming that $M > n$.

Observe that the dominating time bound above is $O(M \log M)$. Thus we have that our algorithm runs in time $O(M \log M)$. So we have given the algorithm and shown the time bound for it. Now we will show correctness. Consider the polynomial $P(X)$ as seen in item 2. Let D_P be the list of the degrees of the terms of $P(X)$. For example, for $P(X) = x + x^7$, $D_P = 1, 7$. Then we have that D_P contains every possible sum of one element from A and one element from B, except with each sum shifted up by $2M$. But then we can just shift each number in the list down by $2M$ so that it actually contains every possible sum, which is exactly what we do in our algorithm. So let D_P be this newly shifted down list. Then since $L = D_P$, we have that L contains every possible $a_i + b_j$.

Therefore, we need only to know if any element from L plus any element from C sums to 0. That is, each must check if there exists i, j such that $l_i + c_j = 0$, $l_i \in L$. Our algorithm does exactly this, checking if if there exists an $l_i = -c_j$ for every $c_j \in C$. Thus it follows that our algorithm works correctly.

Problem 4, CS38 Midterm, Matt Lim

We will use the term "weight of a triangle" to indicate the sum of all three of its edges. Now, here is pseudo-code for our algorithm.

Lightest-Triangle(P: set of n points in the plane)

1. sort by x coordinate and split equally into **L** and **R** subsets
2. (a, b, c) = **Lightest-Triangle(L)**
3. (d, e, f) = **Lightest-Triangle(R)**
4. $D_1 = d(a, b) + d(b, c) + d(c, a)$
5. $D_2 = d(d, e) + d(e, f) + d(f, d)$
6. $D = min(D_1, D_2)$
7. scan **P** by x coordinate to find **M**: points within $D/2$ of midline
8. sort **M** by y coordinate
9. Starting at the point with the highest y coordinate (call it v), compute lightest triangle containing v using all the points within 24 positions ahead in the sorted list. Then iterate through the sorted list **M** and do this for every other point. Call the lightest triangle found by this iteration (g, h, i).
10. return lightest among (a, b, c), (d, e, f), and (g, h, i).

We will now prove time bounds for this theorem. We have that the time is

$$T(n) = 2T(n/2) + \text{ time for middle } + O(n \log n)$$

where the $O(n \log n)$ is the time it takes to perform the splits and sorts. Our goal is to show that the time for middle is $O(n \log n)$. To do this, we need only to look at our algorithm. For each point in the middle area, we compare at most $\sum_{i=1}^{24} i$ triangles. This is a constant number. And since the number of points in the middle area is at most n, we have that the middle area takes linear time. So then we get that

$$T(n) = 2 \cdot T(n/2) + O(n \log n)$$

Then by the proof shown in class (lecture 7, slide 42) we have that

$$T(n) \leq c \cdot n \cdot \log^2 n$$

So we have that our algorithm runs in time $O(n \log^2 n)$.

Now we will prove correctness. To do this, it suffices to show that we compute triangles across the middle correctly. So, let us consider the middle area; that is, the area composed within distance $D/2$ of the midline (making the total area width D). We only have to consider this area because points outside this area will have a distance $> D/2$ with points across the midline, which by the triangle inequality means that any triangle formed by such points will have a weight $> D$. So, we continue. We will divide this area up into $D/4 \times D/4$ boxes. Now we can make the following observations. First, we have that no 3 points lie in the same box. If so, then they would form a triangle of weight less than D, which is impossible due to how we set things up. Next, we have that if 2 points are within ≥ 24 positions of each other in list sorted by y coordinate, then they must be separated by ≥ 2 rows. Then we have that their distance is $> D/2$. Then by the triangle inequality, any triangle containing these two points must have a weight $> D$. So we don't need to consider these points. In other words, the only possible points that we must consider when going down the sorted list are the points within 24 positions of each other. And since we consider points in order of highest to lowest y coordinate, we only need to consider triangles that can be formed with points within 24 positions ahead of each point. So we consider every relevant triangle across the middle and pick the lightest one. Thus it follows that our algorithm is correct, since we compute the lightest triangle of the left half, right half, and middle correctly, and take the minimum.

Problem 5, CS38 Midterm, Matt Lim

Let x be some set vertex in G. We will let it be v_k for some $1 \leq k \leq n$. Let $HC(S, v)$ be 0 if there is no path from x to v that passes through exactly the set S and 1 if there is such a path. We will organize our subsets of vertices in the table in the following way. $S_1, S_2, \ldots, S_n$ will be the subsets containing only a single vertex, with S_1 containing v_1, S_2, containing v_2, etc. Next will come the subsets containing two vertices, then three, and so on. This means S_{2^n-1} is the full set of vertices. Here is pseudo-code for our algorithm.

Ham-cycle(G)

1. Test G to see if it is a strong connected component. If yes, continue. If not, return false.
2. for $i = 1$ to n
    3. for $S = S_1$ to S_n
        4. if $v_k \in S$
            5. $HC(S, v_i) = 1$
        6. else
            7. $HC(S, v_i) = 0$
8. for $i = 1$ to n
    9. for $S = S_{n+1}$ to S_{2^n-1}
        10. $HC(S, v_i) = 0$
        11. if $v_i \in S$
            12. for each edge $(u, v_i) \in E$ (if multiple such edges exist only consider one of them)
                13. if $v_i \neq v_k$ and $HC(S - v_i, u) == 1$
                    14. $HC(S, v_i) = 1$, break from loop
                15. elif $v_i == v_k$ and $HC(S, u) == 1$
                    16. $HC(S, v_i) = 1$, break from loop
17. if $HC(S_{2^n-1}, v_k) == 1$
    18. return $true$
19. return $false$

We will now show that this algorithm runs in time $O(n^2 2^n)$. Line 1 takes $O(n + m)$ time, as seen in lecture 2. Lines 2-7 take time $O(n^2)$, since each run through the loop is constant time. This leaves lines 8-17. We can see that this builds a table of size order $n \cdot 2^n$. For each cell of the table, we do order n work, as we must look at all the neighbors of the current vertex, of which their can be at most n. So overall, these lines take $O(n^2 2^n)$ time, and we can see that the whole algorithm takes this time as well.

We will now show correctness. First, note that we test the graph to see if it is strongly connected. This is because if there is a Hamilton cycle in G, then G must be strongly connected. This is because if there is a Hamilton cycle, every vertex is reachable from every other vertex through the Hamilton cycle. Now, onto the rest of the algorithm. Note that each subproblem (S, v) is to determine if there is a path starting at x that passes through exactly S and ends at v. The culmination of caching the results of all the subproblems is to determine whether $HC(S_{2^n-1}, v_k)$ is 1, which would mean that there is a path from v_k to itself (a cycle) that passes through all the vertices. So, let us explain how this table gets use the correct result for this cell. We have that for a set of n vertices, there are 2^n subsets of those vertices, including the null set. This is why we have $2^n - 1$ columns (we don't include the null set). For each subset S, we want to know if there is a path from x to v that passes through exactly S for every vertex

v in the graph (if we start and end at x, then "exactly S" means that we start at x, pass through every element in $S \backslash x$, and end at x). For lines 2-7, we only consider subsets with one element. And since we start at vertex x, that means that the only cell that will be filled with a one will be the one indexed by $(\{x\}, x)$. This is our initialization of the table. Next comes the dynamic programming part. Let us examine lines 11-17. For each cell, we do the following. If $v_i \notin S$, then the path cannot contain exactly S. So we can just mark the cell with a 0. But if $v_i \in S$ and $v_i \neq v_k$, then we do the following. We consider all vertices with edges that are directed towards v_i. That is, we consider all vertices that can get to v_i through a single edge. Then, for each of these vertices, we consider if we can get to it from x, passing through $S - v_i$. If we can, then we can clearly get to v_i from x passing through S. We do something slightly different if $v_i \in S$ but $v_i == v_k$. We again consider all vertices that can get to v_i through a single edge. However, for each of these vertices (we will call each one u_k), we consider if we can get to u_k from x passing through S. Notice that we do not alter S here. This is because v_i is our starting vertex, and any paths from x to u_k must include v_i itself. Thus we do not want to subtract it out of S. Then it is clear that, if there is a path from x to some u_k that passes through exactly S, then there is a path from x back to $v_i = x = v_k$ that passes through exactly S as well (since here we do not double count starting/finishing at x). So we have gone through all the cases for each cell. Now we can see that, by building this table up cell by cell, we get exactly the right answer for each subproblem. And eventually, all we will have to do is look at each vertex that can get to x using one edge and see if any are reachable from x going through S_{2^n-1}. This will give us the correct value for $HC(S_{2^n-1}, x)$ and thus the correct answer to our problem. Thus we can conclude that our algorithm works correctly.