Problem 2, exam 1

Here's Prim's algorithm, modified slightly to use C syntax.

```
MSTPrim (G, w, r):

Q = V[G];
for (each u ∈ Q) {
   key[u] = ∞;
}
key[r] = 0;
π[r] = 0;
while (Q not empty) {
   u = ExtractMin (Q);
   for (each v ∈ Adj[u]) {
      if (v ∈ Q && w(u,v) < key[v]) {
         π[v] = u;
         key[v] = w(u,v);
      }
   }
}
```

Part a asked you to give an estimate of the worst-case running time on a graph with n vertices and e edges. The priority queue is used in five places: it's initialized in the assignment to Q, it's checked in the while condition, the vertex u is extracted from it, a value of one of its elements v may be updated as a result of relaxation, and v is tested for membership in Q.

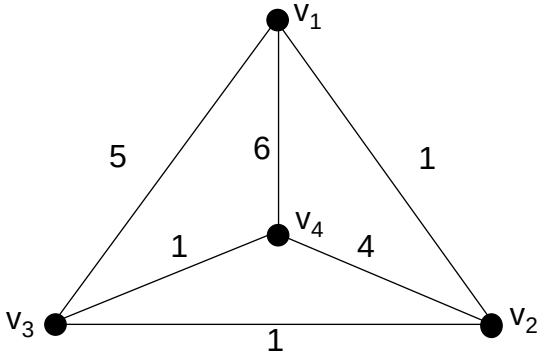
Initialization of the queue is $O(n)$ and checking for an empty queue is $O(1)$. There are two places that using an unsorted linked list affects the running time: ExtractMin, where *all elements* must be checked to determine the element with smallest value, and updating an element, which may require movement in a binary heap but requires no extra processing in an unsorted list. The test for membership in the queue can be implemented merely by a test of a vertex's pointer into the queue; a null pointer means that the vertex is not queued. A less efficient implementation is to search the queue, which requires $O(n)$ operations. This would not be any less efficient than the corresponding operation using a binary heap, however.

At any rate, CLR's analysis noted that the outer loop in each algorithm executed once for each vertex, while the inner loop was executed once for each edge. Thus ExtractMin contributes $n \times O(n)$ operations, and updating contributes $e \times O(1)$ for a total of $O(n^2 + e) = O(n^2)$ in each algorithm. The membership test in Prim's algorithm, if coded inappropriately, contributes $e \times O(n)$ operations, which dominates the total running time.

Interestingly, ExtractMin exhibits worst-case behavior with *any* assignment of weights to edges, since every element in the queue must be examined to find the minimum. A more detailed analysis, however, reveals that the number of updates will depend on the assignment of edge weights; in the worst case, a neighbor's key value will be updated every time the neighbor is examined. A pattern that produces

the worst case and an example of a corresponding assignment of edge weights are shown below.

vertex	*action*
v_1	update v_2, v_3, and v_4
v_2	update v_3 and v_4 each for the second time
v_3	update v_4 for the third time



Problem 3, exam 1

This problem involved getting a Θ value for the solution to the recurrence

$$T(n) = 9T\left(\frac{n}{3}\right) + n^2 + n \lg n$$

The recurrence can be solved with the Master Theorem. The variables a, b, and f have values a = 9, b = 3, and $f(n) = n^2 + n \lg n$, which is $\Theta(n^2)$. The value of $\log_b a$ is 2 since $9 = 3^2$. Case 2 of the Master Theorem applies to give an estimate of $\Theta(n^2 \lg n)$. An alternate form of the Master Theorem, given on page 9 of the Readings, uses a d variable, which is 2 in the recurrence. Using this version of the Master Theorem was fine.

With difficulty, one could produce a solution using the iteration method instead of the Master Theorem.

Problem 4, exam 1

You were to prove that a directed graph G = (V, E) with no isolated vertices is strongly connected if and only if there is a cycle in G that includes every edge at least once (and possibly more than once). As with any "if and only if" proposition, there were two things to prove:

1. If G is strongly connected, then there is a cycle that contains every edge at least once.
2. If there is a cycle that contains every edge at least once, then G is strongly connected.

The second part is the easier of the two. Since the assumed cycle contains every edge and there are no isolated vertices, the cycle also contains every vertex. Consider any two vertices u and v. A path from u to v is found by starting at u on the cycle and following it around to v; a path from v to u is found in the same way. These two paths,

found for all pairs of vertices in the graph, satisfy the conditions for strong connectivity.

There were two successful approaches to the first part:

- a proof by contradiction that involved assuming that no cycle could include every edge;
- a proof by construction, involving the repeated augmenting of a cycle to include more edges.

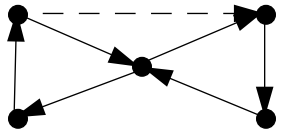
Here are sample solutions for each approach.

Suppose that G is strongly connected, yet no cycle includes every edge in the graph. Let C be a cycle that includes the maximum number of edges of G, and let (u,v) be an edge not included. Let w be some vertex on the cycle. Since G is strongly connected, there is a path P_1 from w to u and a path P_2 from v to w. These two paths, along with (u,v) and C, comprise a longer cycle. It starts somewhere on C, reaches w, detours on P_1 through (u,v), returns to w on P_2, and continues along C to the starting point. It also contains all the edges in C plus an edge not in C, namely (u,v), which contradicts the assumption that C included the maximum number of edges possible. Thus there is no cycle that includes as many edges as possible without including them all, which is what we wanted to prove.

A proof by construction is similar in many respects. (It is similar in addition to the Euler tour proof on an early homework assignment.) The construction starts with some cycle C_1 that, say, contains a vertex w. For each edge (u,v) that's not on C_1, we find the path from w to u and the path from v to w; this cycle, added to C_1 in the same way as described in the previous proof, creates a cycle C_2. Repeat this process until some C_k includes all the edges in G.

Another constructive proof gradually increases a path until it includes all the edges, then finds a way back to where it started. This approach picks a starting vertex, finds an edge (u,v) that's not yet traversed, and travels to u and then across the edge. Strong connectivity provides the path to u. From v, the algorithm finds another untraversed edge, and extends the path through that edge in the same way. This process continues until all edges have been traversed, at which point it finds a path back to the start vertex provided by the strong connectivity assumption.

Few solutions were completely correct. One false claim was that one of the endpoints of an edge not on a cycle was unreachable from the cycle. Another was that since all vertices u and v are connected by paths from u to v and back, the collection of cycles that results covers *all* the edges of G. The graph below shows a counterexample in which the dotted edge is not in any of a particular set of cycles.



Problem 5, exam 1

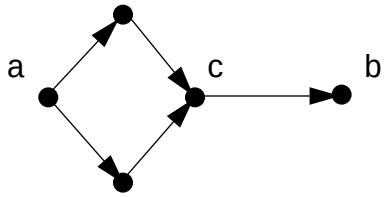
You were to find an efficient algorithm that, given a directed acyclic graph G = (V, E) and a vertex a in V, counts the number of paths from a to each other vertex. (The original problem said "all other vertices", but this was corrected at the exam. Another clarification made at the exam was that G contains no multiple edges; that is, for any u and v, the number of edges connecting u to v is at most 1, and the number of edges connecting v to u is at most 1.)

There were a number of solution approaches. The best one took advantage of the fact that G is a DAG and can therefore be topologically sorted. This leads to an algorithm that runs in time O(n+e):

> Set the path counts of all vertices to 0, and set a's path count to 1.
> Topologically sort the vertices.
> For each vertex u in topological order, do the following:
> For each neighbor v of u, add u's path count to v's path count.

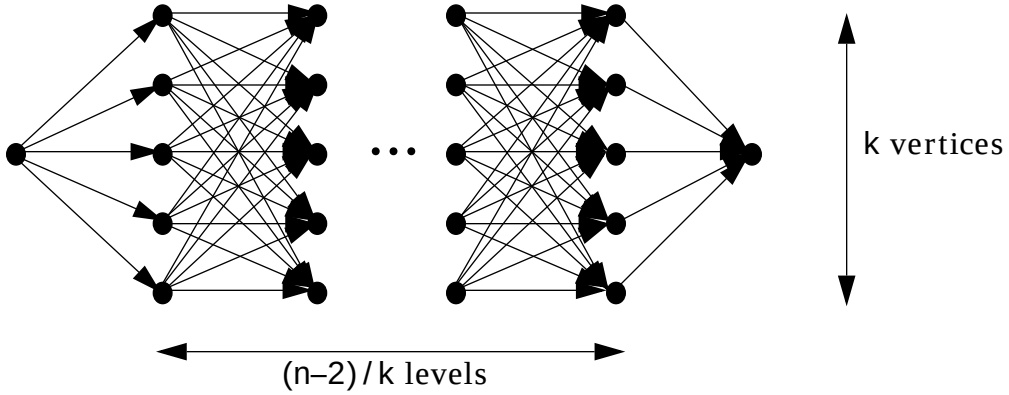
To prove this, one would note that a vertex u isn't reached until all the path counts of vertices between it and a have been updated; this is because of the topological ordering. Thus, when u is processed in the outer loop, the number of paths to each of its predecessors is known and tallied in u's own count.

Not taking account of multiple ways to get to a vertex from a was the flaw in another common solution, namely applying CLR's DFS or BFS algorithm directly and merely increment a vertex's path count each time it is visited. The problem with this is demonstrated in the graph below. The first time vertex c is encountered, it's marked;



when c is next encountered on the second path out of a, there will be no way to go past it to add the second path into b's count.

A third approach attempted to solve the problem of missing paths by continuing the search through marked vertices. This search—again, either depth first or breadth first—traverses *every* path from a to every other vertex. Unfortunately, the tradeoff is speed; this algorithm run in time *exponential* in the number of vertices. An example on which it performs badly is shown below. If k is approximately equal to the



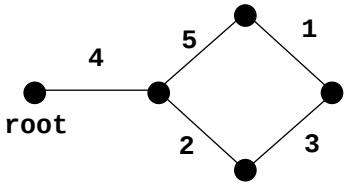
square root of n, the number of paths from one end of the graph to the other is approximately $n^{(\sqrt{n})/2}$.

One last approach arose from noting that the (u,v)-th entry in the kth power of the adjacency matrix of G is the number of paths of length k from u to v. If the first n–1 powers of G's adjacency matrix are added, the resulting matrix contains the desired path counts. Matrix multiplication (at least the standard algorithm) is $O(n^3)$, and there are n–1 multiplications, yielding a runtime estimate of $O(n^4)$.

Problem 4, final exam

Part a

The lightest edge might not be in a shortest path tree, for example:



Part b

The shortest path might change. Suppose there are two shortest paths, one with k edges and one with k+1 edges. After the increase, the path with k edges is now shorter than the path with k+1 edges.

Problem 5, final exam

Solution: Root the tree, then compare the set of vertices on levels 0, 2, 4, … with the set of vertices on levels 1, 3, 5, … At least one of these sets must contain at least half the vertices in the tree.

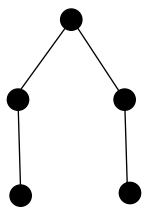
Another approach uses a proof by induction. The base cases are a single vertex and a single edge. In the induction step, we consider two cases: either the tree has an odd number of vertices or it has an even number of vertices. If the tree has an even number of vertices, we remove a leaf and find an independent set of size ceiling((n-1)/2) vertices in the tree that remains. Since ceiling((n-1)/2) = ceiling(n/2) when n is even, this set qualifies. If the tree has an odd number of vertices, we need the following lemma:

- Any tree with at least three vertices has a leaf with no sibling or has a leaf with another leaf for a sibling. Proof is by induction. All trees of size 3 and 4 qualify. From a larger graph, remove a leaf and find the designated leaf in the result; either it qualifies as the designated leaf in the larger graph, or it is the parent of the leaf that was removed, which itself qualifies as the designated leaf, or it is a sibling of the parent of the leaf that was removed, in which case the removed leaf also qualifies as the designated leaf.

Now the tree with an even number of vertices must contain a leaf x that either has 0 siblings or has a leaf for a sibling. In the first case, we remove x and its parent from the tree and find an independent set of size (n-2)/2, then we add x to that set. In the second case, we find the independent set of size ceiling(n-1)/2 that results from removing x. If x's parent is not in the set, we add x; otherwise we replace x's parent by x and its sibling.

A proof based on creating the desired set out of leaves goes as follows. Suppose there are k leaves in the tree: include all of them in the independent set. Then remove the leaves and their parents. (This may disconnect the tree and create a forest.) The number of leaves is at least as large as the number of parents, so the remaining trees collectively have $n' \geq n-2k$ vertices. Any isolated vertices were formerly attached to the parents, not the leaves, so they may also be included in the independent set. Now find independent sets in all the individual trees; this produces a set of at least ceiling((n-2k)/2) vertices. Then put the k leaves back in.

Note that the leaves alone may not provide the desired set. Consider, for example, the following tree:



Problem 6, final exam

The algorithm involves two steps. First, verify that G' is a tree, either by counting its edges—there must be exactly $|V|-1$—or by doing a depth-first search to make sure G' contains no cycles. (These both work since G' is connected.) Then make sure every edge in G' is also in G.

G' has to be in adjacency list format or edge list format to make the edge counting fast. G' has to be in adjacency list format to make cycle checking fast. G has to be in adjacency matrix format to make the edge checking fast.

Counting the edges is $O(|E'|)$; verifying containment is $O(|E'|)$ since checking for an edge in G takes constant time; the depth-first search is $O(|E'|)$. (Note that $|E'| = |V|$, at least after the edge count is verified.)

Problem 9 (parts a and b), final exam

Part a

Given S and F and k, is there a hitting set of size at most k? (You were allowed to say "size equal to k".)

Part b

The certificate for showing that the minimum hitting set is in NP is the hitting set itself; we check the hitting set's size, then verify (in at worst $O(n^3)$ time) that it shares at least one element with all the sets in F.

Now pick a graph G in which to find vertex cover of size at most k. Consider the set S that contains all the vertices of G, and the set $F = \{F_j\}$ such that $F_j = \{u,v\}$ for each edge (u,v). (F can be easily computed.) There is a vertex cover of size at most k in G if and only if there is a hitting set of size at most k for F. Proof:

=>
Assume G has a vertex cover of size at most k. For every edge (u,v), either u is in the cover or v is in the cover. Each F_j contains the endpoints of an edge, so for each F_j, whichever of the endpoints is in the cover also hits F_j.

<=
Assume F has a hitting set H of size at most k. Since at least one item from each F_j appears in H, the set {u,v} is hit, so either u is in H or v is in H. But then either the edge {u,v} is covered by including u or v (whichever is in H) in the vertex cover, thus producing a cover of the same size as the hitting set.