Homework 1:
Problem 1:
Place each security guard as far "right" as possible to be able to guard the left most unguarded piece of art. Guard 1 is placed at x_0 + 1. Let g_k be the last placed guard and let x_i be the smallest value such that x_i > g_k + 1. Place g_(k+1) at x_i + 1.

Proof: Let O be an optimal solution and let G be the greedy solution. Assume that O and G place the first k-1 guards at the exact same spots but place guard k at different spots. Note that G places guard k as far to the right of the next unguarded piece of art as possible so G places guard k to the right of O's placement. We can then safely move guard k in O to the right until it matches G's placement. All art that k is guarding before being moved is still be guarded by k after the move. The number of guards used by O is unchanged and every piece of art is still guarded. Running time is O(n) because we traverse the points X in order (if the student assumes the painting positions are given in order). The running time is O(n log n) if the student assumes they have to sort the painting positions.

Problem 2:
Let T[k] store the maximum profit possible after k weeks.
T[k] = max { T[k-1] + l_k, T[k-2] + h_k }.
T[0] = 0 and T[-1] = 0 or -infinity, depending on if you want to allow a high stress job on week 1. The solution max profit is at T[n], and to get the schedule, we work backwards from k=n. If T[k] = T[k-1] + l_{k} then we did the low paying job at time k and set k:=k-1. Otherwise we did the high paying job at time k and no job at time k-1, and we set k:=k-2 and repeat.

Proof: Assume T[k-1] and T[k-2] store the maximum profit possible after week k-1 and week k-2 respectively. We have two options on week k. If we take the high stress job, we can't work week k-1. Since it does not matter what jobs we work before week k-1, we should maximize the profit, and by the induction hypothesis, T[k-2] holds this value. If we are working a low stress job, then it does not matter what jobs we do before this job so we should maximize the profit possible. By the induction hypothesis, T[k-1] holds this value.

The running time is O(n) since the table has n entries and it takes O(1) to calculate each entry.

Problem 3:
Using divide and conquer, I split the n teams into two and create a round robin schedule for each. Since there is no overlap in teams, I combine each day of the first half schedule with each day of the second half schedule. Then, I "rotate" the teams of the first group with the teams of the second group, matching each up. Each step of the rotation produces n/2 games that can be played simultaneously so I schedule them all on the same day.

Proof: Assume we can correctly round robin schedule for fewer than n teams. Consider n teams split into two even-ish groups. The "rotation" correctly schedules teams in the first set against all teams in the second set in a minimal number of days. All that is left is to correctly schedule the games for teams within each set, and by the induction hypothesis, we do that.

The running time is the recurrence T(n) = 2T(n/2) + n^2 = O(n^2).

Problem 4:
Using divide-and-prune, let L1=L2=0 and H1=H2=n. Repeat, let M1=(L1 + H1)/2 and M2=(L2+H2)/2, and we request the M1-th largest element of database 1 and the M2-th largest element of database 2. Let a1 be the element returned from the first and a_2 the element returned from the second. Suppose (w.l.o.g.) a_1 =< a_2, if n is odd, we set L1=M1 and H2=M2. If n is even, we set L1=M1+1 and H2=M2. Then we repeat.

Proof: Assume the algorithm correctly returns the median element two databases with H2'-L2' = H1'-L1' < n elements each where the indeces start at L1' and L2' respectively. Given two databases with n elements each and indices starting at L1 and L2, request the (n/2)th element of each. Suppose (w.l.o.g.) that a_1 <= a_2. If n is even then there are at most n-2 elements less than a_1 so a_1 can't be the median. There are also at most n elements larger than a_2 so no element larger than a_2 can be the median. We set L1=M1+1 since no element of database <= a_1 can be the median and H2=M2 because no element larger than a_2 in database 2 can be the median. Now both databases have the same number of element less than n. Since we remove

the same number of elements above and below the median, the median of this smaller database is the median of the whole. So we repeat and by induction hypotheses, we get the median. If n is odd, then there are at most n-1 elements smaller than a_1 and at most n-1 elements larger than a_2. Thus no element smaller than a_1 or larger than a_2 can be the median. We set L1=M1 and H2=M2, and we remove the same number of elements from both databases. By the same induction as before, the median of the recursive call is the median of the whole, and by the induction hypothesis that is what we get.
Problem 5:
Let T[i, d] store the maximum number of events we can photograph if the telescope is at location d at time i.
Set T[0,0] = 0 and T[0,k] = -infinity for all k <> 0; T[k, 86] = -infinity for all k, and T[k, -86] = -infinity for all k
T[i, d] = max {T[i-1,d+1], T[i-1,d-1], T[i-l,d] + {1 if there is an event at location d and time i-1, 0 otherwise}}
The solution is in T[n, d*] where d* is the location of the last celestial event. To get the schedule, we work backwards from T[n.d*]. For each position T[i,d], if T[i-1,d+1] or T[i-1,d-1] equaled to T[i,d], then we must have moved the telescope to this location, otherwise we took a picture at this time.
Proof: Assume T[i-1,d-1], T[i-1,d+1], and T[i-1,d] store the maximum number of events we can shoot by minute i-1 and locations d, d-1, and d+1. We have three options. Move the telescope up, move it down, and stay and take a picture. Taking the picture gives as an event only if one is occuring where and when we are pointing. Because only the current location of the telescope matters and now what pictures we may or may not have taken before, it makes no sense to take less than the max number possible. By the induction hypothesis, T[i-1,d-1], T[i-1,d+1], and T[i-1,d] store these values.
Problem 6: The algorithm is correct. For grading this one, you can assume the algorithm and running time points are automatically achieved if they are proving the algorithm correct.
Proof: Assume there is an optimal truck loading scheme O. Assume our greedy does the same as O for the first k boxes, but on box k, we do something different. The two options are G sends the truck after loading k when O did not or O sends the truck after loading k but G does not. The first is not possible since package k+1 will not fit in the truck. So O sends the truck. Let us create O' where we do the same as O but load box k into the truck instead of sending it. That removed box k from the next truck reducing the load of that truck. Therefore O' uses the same number of trucks as O, no truck is overloaded, and G and O' now do the same on boxes up to and including k.

Homework 2:
Problem 1:
Create a set {1,....n} corresponding to all the indeces of the array. Let x be the bits of the missing number. x starts out as empty. Let k=log n. Request the kth bit of each entry at an index in the set. If there are more 1 bits returned than 0's, then the missing number starts with a 0. Append 0 to x and remove from the set all those indeces that returned a 1. Otherwise is starts with a 1 and append 1 to x and remove from the set all those indeces that returned a 0. Decrement k. If k=0 x stores the missing number.

The total number of steps is O(n) in the first round, O(n/2) in the second round and so on. This is O(n + n/2 + n/4 + ...) = O(n).

Proof: The base case is when n=1. There will be a single bit, and if we query it we see what is missing. Assume we can find the missing number from a set of k digits numbers. Consider a set of k+1 digit numbers. If the set is even is even there should be the same number of entries that start with a 0 as start with a 1. If the set is odd, there should be one more entry that starts with a 0 as starts with a 1. If there are more 1's than 0's, then we know that the missing number starts with a 0. Otherwise it must start with a 1. We remove all those indeces from the set that corresponds to the majority answer (what the missing number cannot be). Then we have effectively looking at a smaller set of (k-1) digit numbers. By the induction hypothesis, we find the missing number.

Problem 2:
Sort the pizzas by value so that v_1 >= v_2 >= .... >= v_n. Then run through the list in order. For pizza i, we schedule it as late as possible and still meet the deadline. So we go to the deadline d_i and work from there to time 0 and schedule the pizza in the first available slot. If there is none available, we schedule the pizza as late as possible in the schedule. So we go to time n and work backwards scheduling it in the first available slot.

The runtime is O(n log n) to sort, O(n) to run through the list, but at worst case I may have to run through O(n) slots before finding a free slot. For a total run time of O(n^2).

Proof: Assume we have an optimal schedule O, and if we consider the pizzas in the order scheduled by greedy, O and G schedule the first k pizzas the same but differ on the time for pizza k+1. Let pizza x be scheduled in O at the time when G schedules k+1. Create schedule O* where O and O* are the same except that O* swaps the scheduling of pizzas x and k+1. O* now matches G on k+1 pizzas, but we have to prove that O* is just as good a schedule as O. Here are the cases:

Case 1: O schedules k+1 before G/O* does.

Case 1a: O schedules k+1 before its deadline but O* does not. This is impossible since G will schedule k+1 before the deadline if possible.

Case 1b: O and G/O* both schedule k+1 after the deadline. O and O* still pay the penalty for pizza k+1, but pizza x is moved earlier on O* than on O. So at worst O* pays the same penalty as O.

Case 1c: O and G/O* both schedule k+1 before the deadline. Neither O nor O* pays a penalty for pizza k+1, and pizza x is moved earlier in the schedule in O* than in O. So at worst O* pays the same penalty as O.

Case 2: O schedules k+1 after G/O* does.

Case 2a: O schedules k+1 before its deadline. This is impossible. If O schedules k+1 before its deadline, there is an available slot before its deadline, and G put k+1 in the last available slot before its deadline.

Case 2b: O schedules k+1 after its deadline. Since G puts k+1 in the last available slot, G (and therefore O*) must put k+1 before its deadline. Now in O*, pizza x is scheduled later than in O. So O* may have to pay the penalty on pizza x if it now misses its deadline. However, O pays the penalty of pizza k+1. So penalty(O*) <= penalty(O) - v_{k+1} + v_x. Since v_{k+1} >= v_x, we have penalty(O*) <= penalty(O).

Problem 3:
Let T[i,j] be the maximum profit we can earn if we do j buys and sells and the last sell occurred at time i or earlier. We let T[i,0] = 0 for all i,
T[i,j] = max{T[i-1,j], max{T[x, j-1] + 1000(p(x+1)-p(i))} for all 2(j-1)<=x< i}

Prove that T[i,j] is the maximum profit from making j buys and sells that complete by time i. Assume T[i-1,j] stores the maximum profit from making j buys by time i-1 and T[x,j-1] stores the maximum profit from making j-1 buys by time x for all times less than i. We have two choices. If we do not complete a sale at time i, then the best we can do is the maximum profit at time i-1, and by the induction hypothesis, T[i-1,j] stores this. If we complete a sale at time i, then we purchased those shares at time some time x+1, and we must have completed the previous j-1 sale at some time at or before x. Since we are buying at time x+1, the want the maximum profit possible at time x, and by the induction hypothesis T[x,j-1] stores this value. We take the maximum over all these choices.

Here is how to get the solution from the table. This will be needed in the homework, but don't deduct points if the students forget this in the quiz. The solution is the largest value T[n,m] for all m from 0 to k. We then can work backwards from here. Start with s=n and t=k. From T[s,t], if T[s,t] = T[s-1,t] then we did not complete a sale at time t. Otherwise, we go search for all x from 2(t-1) to s for the T[x,t-1] that stores T[s,t] - 1000(p(x+1,t-1), and that means we bought at time x+1 and sold at time t. We then set s=x and t=t-1 and repeat. (Another solution is to have a second table that stores the choices made at each step. If T[i,j] = T[i-1,j], the set S[i,j] = 0 to indicate no sale. Otherwise, set S[i,j] = x+1 to record that we bought at time x+1 and sold at time j.

The size of the table is n x k, and it takes up to time n to go back to find each value for a running time of O(n^2 k).

Problem 4:
Consider the x_i values from smallest to largest, and match each x_i to the interval t_j that overlaps with x_i and, of those that overlap, has the smallest t_j + \epsilon_j.

This algorithm is O(n^2) because for each of the n times, we may have to run through n intervals to test for overlap and to store the minimum end time.

Proof: Let G be the matching our greedy algorithm produces and let O be the matching that some optimal algorithm produces. Consider the values from smallest to largest and suppose that G and O match the first k-1 values to the same intervals, but now assume that G matches the x_k to interval t_a while O matches x_k to interval t_b and O matches some other value x_p to t_a. Now we have the following:

t_b - \epsilon_b <= x_k	because O matches x_k to t_b
<= x_p	because G considers the values in order
<= t_a + \epsilon_a	because O matches x_p to t_a
<= t_b + \epsilon_b	because G matched x_k to t_a instead of t_b

Therefore, we can create O' by matching x_k to t_a instead of t_b and matching x_p to t_b instead of t_a. Since all other matchings stayed the same, O' has the same number of matchings as O and it agrees with G on the first k values.

Problem 5:
Let T[i, j] = true if the first i+j characters of s is an interleaving of i characters of s1^p and j characters of s2^q for any non-negative p and q.

We have two choices, either match the next character of s with s_1 or match the next character of s with s_2. The recurrence is:
T[0,0] = true
T[i, j] = (T[i-1, j] && s[i+j-1] == s1[(i-1) % s1.length]) || (T[i, j-1] && s[i+j] == s2[(j-1) % s2.length])

(s[x] means the x'th character of string s, and I am assuming the string index starts at 0 like in C or Java.)

Let n be the length of s. To get a solution, we look at all T[x,y] where x+y = n and see if any are true. The running time is O(n^2) since we have an n x n table and it takes O(1) to fill in each entry.

Proof. Assume T[i-1, j] indicates whether the first i+j-1 characters of s is an interleaving of i-1 characters from s1^p and j characters from s2^q, and assume T[i,j-1] indicates whether the first i+j-1 characters of s is an interleaving of i characters from s1^p and j-1 characters from s2^q. Are the first i+j characters of s an interleaving of i characters from s1 and j characters from s2? If so, then the (i+j)th character is either from s1 or from s_2. If it is from s1, then it must match the ((i-1) % s1.length)+1 character of s1 and the remaining i+j-1 characters are an interleaving of i-1 chars from s1^p for some p and j chars from s2^q for some q. By the induction hypothesis, T[i-1,j] stores this value. The same reasoning applies if the (i+j)th character is from s2.

Problem 6:
If the set S is only one interval {[a,b]} with height h, then return the skyline [(0,0),(a,h),(b,0),(1,0)].

Otherwise, split the set S in half arbitrarily and recurse on each half. The result of the two recursive calls are the skylines [(0, a_0), (y_1, a_1), ..., (y_p, a_p), (y_{p+1},0)] and [(0, b_0), (z_1,b_1), ..., (z_q, b_q), (z_{q+1}, 0)]. We now merge the two skylines together. The first point of the output is (0, max {a_0,b_0}). We run through both skylines from left to right. Suppose we are considering points (y_i, a_i) and (z_j, b_j).

Case 1: If y_i < z_j, add point (y_i, max {a_i, b_j}) and go to the next point in sequence 1.
Case 2: If y_i > z_j, add point (z_j, max {a_i, b_j}) and go to the next point in sequence 2.
Otherwise: add point (y_i, max {a_i, b_j}) and go to the next point in both sequence 1 and 2.

The result does not have maximal intervals, so we run through the output a second time, and if we have a point (x_i, c_i), (x_{i+1}, c_{i+1}) with c_i = c_{i+1}, we remove the second point. (Technically, we must check to see that this lies inside an interval, so we would need to record for each point whether we added it from skyline 1 or 2, but you can ignore that technicality in the grading).

The proof is induction on the size of S. The base case of S=1 is trivial.

For S > 1, we split the list in two and assume that each half's skyline contains the maximal subintervals of a maximal height. Consider interval [x_i,x_{i+1}] in the output skyline. When the algorithm added the point (x_i,c_i), it chose the maximum of skyline y and z at that point. Under the I.H. that each skyline is maximal, then c_i is the maximal height at this point. Because we removed duplicates, the point (x_{i+1},c_{i+1}) is only included where one of the skyline changes and only if c_{i+1} != c_i. Thus, the interval is a maximal interval with height c_i.

The running time is T(n) = 2T(n/2) + cn and so T(n) is Theta(n log n).

Homework 3:

Problem 1: We reduce the airplane problem to the Star Wars problem. Each airplane will be come a transmission interval. Plane i with arrival time t_i and flight limit d_i will become interval [t_i, t_i + d_i], or more correctly: (t_i + d_i)/2 +/- d_i / 2. Each time at which you can land an airplane will become a reception time: x_i = 3i. The reduction is linear time.

Now we prove that there is a transmission match if and only if we can land every airplane.

(->) If there is a transmission match, then we can place each x_i into some interval t_j +/- epsilon_j. That means landing time x_i falls into the interval [t_j, t_j + d_j] in which plane j will be at our airport and able to land. So every plane gets a landing time.

(<-) For the other direction, assume there is a way to land every plane. That means for each plane j, we landed it at some time X, and X must be between when the plane arrived at the airport t_j and when it ran out of fuel t_j + d_j. Therefore, we matched transmission X/3 uniquely with interval [t_j, t_j + d_j]

Problem 2:

We convert Hamiltoninan Cycle to Longest Path.

Given G, create G' by taking G and adding three new vertices. New vertex a is connected to some arbitrary vertex v of G. New vertex b is connected to every neighbor of a in G, and new vertex z is connected to b. We set K to n+2 and ask if there is a path of n+2 edges in G'. The reduction is linear time since we add three vertices and at most n edges.

(->) Assume G has a Hamiltonian cycle, then G' has a path from a to v, the cycle will return to v from some vertex u, but u is connected to b and then b to z. That gives a path from a to z that is 1 edge (a,v), n-1 edges (v to u), and 2 edges (u to b to z).

(<-) Assume G' has a path of n+2 vertices. Since there are only n+3 vertices in G', the path must hit every vertex, and since a and z are degree 1, the path must start at a and end at z. Therefore, there is a path from a to v to some vertex u to b to c, and we know v to u hits every vertex and there is an edge from u to v. Adding that edge gives us a cycle of n vertices in the original G.

Problem 3:
Given n numbers and a bound T, create n objects. For number x, create an object with weight x and value x. Make the knapsack capacity C=T and the bound B=T. The running time of this reduction is linear in the size of the input because we make one object for each item, and we are using the same numeric values.

(->) Suppose we have a solution to knapsack: a subset of objects whose total value is >= B and total weight is <= C. Because the sum of the values of the objects <= C = T and the sum of the weights of the objects >= B = T, we know that the sum of the values/weights of the objects = T. Thus using those same values for the numbers of the subset gives a subset that sums to T.

(<-) Suppose we have a subset of numbers that sums to T. Choose those same objects for the knapsack, and their value is T >= B and the weight is T <= C.

Problem 4:
Create a network with m+n+2 nodes. There will be nodes s and t. For each of the n regions, create a node. For region i, let node n_i have an edge from s with capacity N_i. Create m nodes for each of the safe places. For safe place j, let node m_j have an edge to t with capacity M_j. For each region node n_i and safe space node m_j, create an edge from n_i to m_j if and only if we can get from region i to safe space j in under 24 hours. Give that edge infinite capacity. Run flow, and the flow from edge i to edge j is the number of people to send from region i to safe space j. The run time is O(nm) to create the network (n+m+2 vertices and at most n+m+nm edges), and by Edmunds-Karp, the flow will take O((n+m)(nm)^2) for a total running time of O(n^2m + nm^2).

(->) Suppose we can evacuate K people safely. Then we can create a flow of size K. Send a flow from s to n_i equal to the number of people leaving that region. Since that number is at most N_i, the flow is under the edge capacity. Send the flow from i to j equal to the number of people who go from region i to safe space j. The capacity of that edge is infinite. The total flow out of node i will equal to flow in. The total flow into node m_j will equal the number of people arriving, and send that flow on to the sink. Since that number is under M_j, it is less than that edge capacity.

(<-) Suppose we have a flow of size K. Then for each edge from n_i to m_j that has flow on it, say that flow is x, send x people from region i to safe space j. Since the total flow into the sink (and thus into the safe space nodes) is equal to K, we successfully moved K people into the safe spaces.

Problem 5:
Using the straight line route and distance d_1, and for each cell tower location t_i, calculate a_i and b_i where a_i is the first point at which cell 1 enters the range of tower i and b_i is the point on the route at which cell 1 leaves the range of tower 1. This can be done in time O(n), Now, create a network. For each cell and each tower create a vertex. Have an edge from cell vertex i to tower vertex j if cell i is in the range of tower j, and give that edge weight 1. Have a vertex start and an edge of capacity 1 from start to each cell vertex, and have a vertex sink and an edge of capacity 1 from each tower vertex to sink. Now run flow. Since the maximum flow is n and since there are at most n^2 edges, by Ford Fulkerson, this flow takes O(n^3) time. Now, move the cell 1. Suppose cell 1 is connected to tower x, it can stay connected to tower x until it reaches point b_x. Now, we remove that 1 flow from source to 1 to tower x to sink, and we change the edges from cell 1 to the towers based on the towers it can now reach. We keep the rest of the graph and flow the same (so we have a flow of n-1). Now we see if we can adjust to reconnect cell 1. This requires a single iteration of the augmenting path to go from a flow of size (n-1) to a flow of size n, and so is done in O(n^2) time. We repeat this process until cell 1 reaches its destination. If there ever is a situation where we can't find an augmenting path to get a flow of n, it is impossible to connect all the cells to the towers. We have to readjust the flow at most n times for a total run time of O(n) + O(n^3) + O(n^3) = O(n^3).

(->) Suppose at each step it is possible to connect all the cells to the towers, then we can create a flow of size n at each point of the path by sending 1 unit of flow to each cell, and that cell sending the unit to the tower it is connected to and from there to the sink.

(<-) Suppose we were able to find a flow of size n at each of the b_x points on the path. Then there is 1 unit of flow going to each cell node and 1 unit from that node to a tower node. Each tower node receives only 1 unit of flow, and so we connect that cell to that tower. Each cell is now connected, and each tower gets 1 cell. We can keep this connection until cell 1 leaves the range of the tower it is connected to, and that only happens at the b_k points.

Problem 6:
Here is a reduction from Vertex Cover. Given G with n vertices and m edges and K as an instance of Vertex Cover, we create a rumor instance as follows. Create n+m actors, one for each vertex and edge of the graph. An actor j follows i if and only if edge j is incident to vertex i. We set k=K and b=m+k. Set p(a) = 1/2 for every actor a. The construction takes O(n+m) time.

(->) If there is a solution to the rumor problem, then we create a set S of the vertices G whose actors are in k planted rumors. This set has at most K elements. Since only "edge" actors follow anyone, the only way to get k+m actors posting the rumor is to have all edges receive the rumor. Thus the vertices of S must cover every edge of G.

(<-) Likewise, if we have a solution of Vertex Cover, then we plant the rumor at the actors that correspond to the vertices of the cover. Since every edge is covered, each of the m "edge" actors' will receive the rumor from one of the two actors it is following and will post the rumor. Thus we end up with m+k actors posting the rumor.

Homework 4:
Problem 1:
Let V = emptyset, and for each vertex v of G, let L[v] = infinity for v != h and L[h] = t_0. Let P[v] = null for all v.

```
Repeat :
   Let v be the vertex of V(G) - V that has the smallest L value.
   Add v to V.
   For each edge (v,a) of G, set L[a] = min{L[a], L[v] + f_{v,a}(L[v])}, and if this changed L[a], we set P[a] = v.
Until (v = w) or until all vertices chosen.
```

We can get the path by following P[w], P[P[w]], P[P[P[w]]], ..., h.

The running time of the algorithm is identical to Dijkstra's. Every edge is considered at most twice, and we have to maintain a sorted list of L. We can use a heap for that and so O(m log n).

Proof: Suppose that the tree of least cost paths from h to all other vertices matches an optimal tree for the first k vertices, in the order our algorithm added them, but then the optimal algorithm uses a different path to vertex k+1. Let x be the vertex immediately preceding vertex (k+1) on the optimal path and y be the vertex immediately preceding (k+1) on the greedy path. Either x < k+1 or x > k+1 (ordered by how they were added to V by greedy). If x < k+1, then by the induction hypothesis, L[x] = the optimal time we can reach O, and then greedy considered L[x] + f(x,k+1) and L[y] + f(y, k+1) and found that L[y] + f(y, k+1) <= L[x] + f(x, k+1). Thus we can change O to have y precede k+1 instead of x, and the time to k+1 can only decrease, and thus the time to any vertex after k+1 also can only decrease. Suppose x > k+1. Then there are two vertices x' and x'' with x' < k+1 and x'' > k+1 where the edge (x',x'') is on the optimal path to k+1. By the same argument as above, we know L[y]+f(y,k+1) <= L[x'] + f(x',x''). By the induction hypothesis, L[x'] is the optimal time to x', and so changing O so the optimal path to k+1 goes from y can only decrease the time to k+1, and thus also only decrease the time to any vertex reached from k+1.

Problem 2:

To prove the problem is NP-hard we do a reduction from Subset Sum.

Given a set of n numbers, {v_1, ..., v_n} and a desired sum S, create a transportation network as follows. For number v_i, create vertices s_i, x_i, y_i, t_i and edges (s_i, x_i), (s_i, y_i), (x_i, t_i), and (y_i, t_i). Each edge has cost and length equal to 1 except that (s_i, x_i) has length v_i + 1 and (s_i,y_i) has cost v_i + 1. We create the full network by having s = s_1, t_i = s_{i+1}, and t_n = t. We set C = 2n + S and we let L be 2n - S + (v_1 + ... + v_n) (any L that is at least this amount will work).

Suppose there exists a path from s to t with total cost C and total length L. Each time the path entered a vertex s_i, it had to choose between going to y_i or going to x_i. If it goes to y_i, we add v_i to our subset Y, and if it goes to x_i we do not. Every path from s to t is exactly 2n edges. Thus the sum of all v_i in Y is C - 2n = S, and we have a solution to Subset Sum.

Suppose we have a subset of {v_1, ..., v_n} that sums to S. Then we choose a path in the network. Each time we get to s_i for some element v_i, if v_i is in the subset, we go to y_i, and if it is not, we go to x_i. The result is a path where C = 2n + S and L = 2n - S + (v_1 + ... + v_n). We have a solution to the path problem.

Problem 3:
Sort the edges of G by bandwidth so that b(e_1) >= b(e_2) >= .... Consider the edges in this order and add the edges to the graph until a and b are in the same component. This algorithm costs O(m log m) to sort the m edges, and we will do a union-find data structure to keep track of the components. This is O(log n) (or O(log* n)) per edge addition. Thus the total running time is O(m log m) = O(m log n) and the answer is the bandwidth of the last edge added.

Suppose there is an optimal path O between a and b that has greater bandwidth than what we get from the algorithm. Let x be this bandwidth. This is a contradiction because the algorithm considered the edges in decreasing bandwidth, and when it added all the edges with bandwidth x or greater in G, a and b were not connected.

Problem 4:
A reduction to Dijkstra. For each edge, if the edge is compromised, give it weight 1 and if not compromised give it weight 0. Use Dijkstra to find the shortest path from s to t. The run time of Dijkstra is O(m log n) and the reduction is O(m) to give O(m log n).

The path uses K compromised edges if and only if the cost of the shortest s - t path in the new graph is K. Assume the path has K compromised edges, then it used K edges of weight 1 and the rest of weight 0. Assume the shortest path is length K, then it used K edges of weight 1, and these are the K compromised edges.

Problem 5:

Create a single source and connect to all the supply nodes and a single sink and connect the distribution nodes to it. Give every edge infinite capacity and every node (except the source and sink) capacity 1. Convert this network to a "normal" flow network, and the result is the "vertex edges" have capacity 1 while the rest have infinite capacity. Run FF and find the minimum cut by doing a DFS/BFS from the source on the residual graph. Since the maximum flow is n (assume n vertices and m edges), the run time is O(nm).

Proof: Assume we have a flow of size K. By the max flow/min cut theorem, there are K edges we can cut, and they must be "vertex" edges. So if we remove those vertices from the original graph, we separate the source nodes from the distribution nodes. (Technically we separate the source from the sink, but the cut can't happen at the source or sink, and so it is either an internal node or a supply/distribution node, but that will still separate the remaining supply nodes from the distribution nodes).

Assume we can separate the graph by removing K nodes. Then in the modified graph, we can also separate the graph by removing those K "vertex" edges. Since each "vertex" edge has capacity 1, there must be a cut of size K.

Problem 6:
First prove such a vertex exists: Let a and b be at distance > n/2, but assume removing no single vertex separates a from b. Therefore, there exists two vertex disjoint paths from a to b. Both paths cannot be > n/2 length, so this contradicts the claim that the distance from a to b is > n/2.

The flow solution is to give every vertex capacity 1 and every edge capacity infinity and to find the minimum cut. The proof is similar to problem 5. Since the max flow is 1, this is an O(m) algorithm where m is the number of edges.

Here is a DFS solution. Run DFS from a. Let v be a vertex on the DFS path from a to b (and neither a nor b). Removing v separates a from b if and only if there is no path from a to b that avoids v. All non-DFS tree edges can only go from a vertex to a direct ancester of that vertex in the tree. So, any a->b path that avoids v must use an edge from the DFS path between a and v to a vertex that is either on the DFS path between v and b or in the subtree rooted at v. Label the vertices from a to b on the DFS where a = 1, the child of a = 2, and so on, until you get to b, and then the DFS tree below b can be labelled arbitrarily. Loop through the labelled vertices in decreasing order starting with the highest labelled vertex. For each vertex w, let shortcut[w] = the lowest label vertex w can reach either directly or going through a vertex with a higher label than w.

```
    directlyreach[w]  = min {a} where (w,a) is an edge
    indirectlyreach[w] = min {shortcut[a]} where (w,a) is an edge and w < a
    canreach[w] = min {directlyreach[w], indirectlyreach[w]}
```

Claim: v separates a from b if and only if for the vertex v+1, canreach[v+1] = v.
Proof: Assume v separates a from b, then the child of v on the path to v cannot be able to reach a vertex above v, so it must have it's "canreach" value

= v. Suppose canreach[v+1] = v. Then for suppose there is a path a -> x -> y -> b where x < v and y > v. Then canreach[v+1] <= canreach[y] < v.

The DFS is O(m), labelling the path is O(n), and calculating canreach considers each edge at most twice, and traversing the path to find a vertex with v = canreach[v+1] is O(n), so the total runtime is O(n).

Homework 5:
Problem 1:
The problem is NP-hard, and we can do a reduction from Subset Sum.

We are given a list of numbers $v_1, ..., v_k$ and a target number *T*. For each number v_i, create a sell order with price 3, quantity v_i secret price 2, and secret min quantity v_i. Create a buy order with price 1, quantity *T*, secrete price 2, and secret min quantity *T*.

There exists a sale if and only if there is a set of numbers that sum to *T*. Suppose there is a sale. Since the buy price is lower than the sell price, the sale has to be at the secret price. Since the buy order is for *T* and the min quantity is *T*, we must have a set of sell orders whose secret min's sum to exactly *T*. Therefore, there is a set of numbers that sums to *T*.

Suppose we have a set of numbers that sum to exactly *T*. If we choose those orders, we have a set of sell orders whose secrete minimums sum to exactly *T*, and that matches the buy order that is requesting a minimum of *T*. Therefore, we have a sale.

The runtime of the reduction is linear time since we are just turning numbers into sales of the same binary value with a little extra constant sized headers.

Problem 2:
Let R[a,b,k] be the best exchange rate of converting from currency a to currency b using intermediate currencies 1,...,k currencies.

Initially R[a,a,0] = 1 and R[a,b,0] = r(a,b) for all a,b.
Either we use currency k as an intermediate currency in the arbitrage or we do not:
    R[a,b,k] = max{R[a,b,k-1], R[a,k,k-1] x R[k,b,k-1]}.

The maximum "cycle" is the largest value R[a,a,n] for all a. To find the cycle, we just run the recurrence backwards. However, this may not be a true cycle because the algorithm does not prevent a vertex from repeating. However, suppose there is a vertex v that repeats twice, then there must exist a v with R[v,v,k] > 1, and so we have found a cycle that is a currency arbitrage > 1, and we can make that trade instead.

The table is n^3 and it takes O(1) to calculate each value for a worst case running time of $O(n^3)$.

Proof: Assume R[a,c,k-1] stores the best we can do trading from a to c using currencies 1 to k-1 for all a and c. To convert a to b using intermediate currencies 1,..,k, we either use currency k, in which case we must convert a to k using currencies 1,...,k-1, and then convert k to b using currencies 1,...,k-1, or we do not use k, in which case we use currencies 1,...,k-1 to convert currency a to b. Because we maximize a product by maximizing each operand, we want the largest currency exchange rate, and by the induction hypothesis, R[a,c,k-1] stores the maximal exchange value for all a and c using currencies 1,...,k-1.

Problem 3:
The problem is NP-hard, and we will do a reduction from Independent Set. We are given an instance of Independent Set: an graph G with n vertices and m edges and a target number K. Create n jobs, one job for each vertex. Divide the day into m distinct time intervals, one for each edge. If a vertex v is incident to edge e, then job v has to do processing during time interval e. Let k = K, and we ask if there are k jobs we can schedule on one processor. The reduction is linear in the number of vertices and edges.

Prove we can schedule k jobs on the same processor if and only if G has an independent set of size K:

Assume we can find k jobs to schedule on one processor. Then there are no overlapping time intervals for any of the k jobs. That means the vertex for a job does not share an edge with any other of the k jobs, and since K = k we have an independent set of size K.

Assume we can find an independent set of size K. No pair of vertices in the independent set share an edge. Therefore none of the jobs associated with the chosen vertices share the same time interval. So we can schedule all the jobs, and there will be no overlap. Since k=K, we are able the schedule k jobs on the same processor.

Problem 4:
The algorithm is: find all Strongly Connected Components (SCCs) of the network. For each edge in a SCC, set its capacity to infinity. For each edge e not in a SCC, set its capacity to m_e. Run the F-F max flow. If there exists an edge e not in a SCC with the flow on e is less than m_e, then we need to push more flow. Run BFS on the network to find an s->t path that uses that edge e, and run BFS from s on the residual graph to find a min cut. There is some edge e* in the min cut that is on the s->t path that includes e. Increase the capacity of that edge e* by 1. Push more flow or find another min cut, and repeat until every non SCC edge has flow >= its capacity. For any edge in a SCC, if the SCC receives flow, then cycle the flow in the SCC until all edges in the SCC achieve their minimum capacity.

[Here are two side points. We are assuming the edges have a positive minimum capacity. If we allow SCC's to have edges of minimum capacity but all edges in and out have 0 minimum capacity, then we can reduce Hamiltonian Path to this problem. We are also assuming every edge with positive minimum capacity is on some s->t path. If not, it is impossible to find an s->t flow that puts flow on that edge.]

Proof: Consider a S-T cut with s in S and t in T. I will call the cut "minimal" if every edge crossing the cut are directed from S to T, none from T to S. Since the flow moves from S -> T across this cut and since there are no reverse edges, any flow that achieves the minimum of every edge in the network must be at least as large as the sum of the capacities of the S -> T edges across the cut. Therefore the flow must be at least as large as the largest minimal S-T cut. Every edge not in a SCC is part of some minimal cut. Assume not, then for every cut that contains this edge has a reverse edge with positive minimum capacity. That would imply that the final flow crosses the cut in both direction, and that implies we have a SCC crossing this cut.

Let e be an edge not in a SCC with flow less than m_e. Consider the s->t path that includes edge e, and let e* be the edge of the min cut that is part of this s->t path. Edge e is not part of any min cut since the flow on e is less than m_e. Therefore there is a larger S-T minimal cut that includes edge e. This minimal cut cannot include e* since there are no back edges in the cut. Thus there exists a larger minimal cut that does not include edge e* so we can freely increase the capacity of e* without affecting the size of this larger minimal cut.

The resulting flow is the smallest flow that achieves flow on every edge. In the preceding round of the algorithm, the min cut found was smaller than the largest minimal cut, and above we showed that the flow must be at least as large as the largest minimal cut. Since each round increases the flow by 1, we have a flow equal to the largest minimal cut.

Runtime: O(m) to find the strongly connected components, then O(f* m) where f* is the capacity of the flow. To compute the flow, at worst we have to do O(m) for one flow augment, then O(m) to find the min cut(s), and repeat until we get to f*.

Problem 5:
The solution uses dynamic programming. Let T[a,b] store the minimum cost of cutting a board of length l_b-l_a that has b-a-1 marks at lengths l_b - l_k for k from a+1 to b.

T[a,a+1] = 0 for all a from 0 to n-1, because there are no marks to cut.
T[a,b] = $\min_{\{k \text{ from } (a+1) \text{ to } (b-1)\}}$ {T[a,k] + T[k,b] + c + k(l_b - l_a)}

The total minimum cost of cutting the board is in T[0,n]. The first cut should be at the kth mark where k is such that T[0,n] = T[0,k] + T[k,n] + c + k l_n. And we repeat for each piece. The optimal place to cut a board that runs from the a-th mark to the b-th mark (of the original board) is at mark k where k is such that T[a,b] = T[a,k] + T[k,b] + c + k(l_b - l_a).

The run time is O(n_3) because the table is nxn (only half the entries used) and we need O(n) to calculate each entry.

Proof. Assume T[a,k] and T[k,b] store the optimal cost of cutting a board of length l_k-l_a at marks a+1 to k-1 and cutting a board of length l_b - l_k at marks k+1 to b-1 for all k from a+1 to b-1. Consider a board of length l_b-l_a with marks at lengths l_b-l_k from k from a+1 to b-1. The first cut can be at any of these marks. Since the total cost is a simple summation of the cost of cutting at each mark, to get the minimum cost we need to optimal the cost of cutting the board after this cut, no matter where we make that first cut. By the induction hypothesis T[a,k] and T[k,b] store the minimum costs for each piece resulting from this cut. Therefore T[a,b] stores the minimum cost of cutting this board.

Problem 6:
(The problem is NP-hard. That is not required, but the proof is a reduction from Subset Sum.)

Here is my approximation algorithm: Sort the packages so w_1 >= w_2 >= ... >= w_n. Consider the packages in order, and place package i into the current truck if it fits. If it does not, send the truck and place package i into a new truck.

Proof. First we show that every truck except the last one ships with more than M/2 weight in it. Suppose a truck ships with less than M/2 in it, and package k was the one we tried to fit and failed. Then package k has weight larger than M/2 which means package k-1 has more than M/2 weight. But package k-1 was on the truck that shipped. That contradicts the assumption that the truck had less than M/2 weight.

The last truck may ship with less than M/2 weight, but the combined weight of the last two trucks is greater than M. If not, then we could have fit the packages in the last truck into the truck before it.

Thus, the average shipping weight of all the trucks is larger than M/2. Since the optimal solution uses K trucks, the total amount of weight shipped is at most KM. The amount of trucks used by the approximation algorithm is less than or equal to the total weight divided by the average weight of each truck. Thus we are using no more than KM / (M / 2) = 2K trucks.