10/10

6.046	Dan Ports
Problem 7-1	
No collab	Recitation #8

Algorithm: We show that it is possible to convert a fractional flow over a graph with integral capacities into an integral flow in $O(VE)$ time. We will do this by considering the residual network of the subgraph containing edges with fractional flow. It is easy to generate this subgraph by iterating through the edges and considering only the ones with fractional flow, then taking the residual network of this subgraph in $O(V+E)$ time. We will show that this residual network necessarily contains at least one cycle. We can therefore apply DFS to find the cycle (or determine that no cycles exist, in which case there are no edges in the residual network and we are done) in $O(V)$ time. Once we have done that, we can traverse the cycle (again in $O(V)$ time) and find the edge with minimum available capacity (call that capacity C). We then augment the original flow with the cycle we have just found, incrementing the flow through each edge by C (or decreasing it if the flow in the original graph is in the opposite direction as the edge in the cycle). As we do so, if any edge now has an integral flow, we remove it from the fractional subgraph and its residual network. We repeat this procedure until the subgraph of edges with fractional flow is empty.

Proof of correctness: To justify the correctness of the algorithm, we first prove the claim that the residual network of the subgraph containing edges with fractional flow contains a cycle (if it is non-empty). To see this, consider any edge e in the fractional subgraph, connecting vertex u to vertex v with maximum capacity c and flow f. Since c is an integer (by definition of the problem) and f is fractional (because e is in the fractional flow subgraph), we know f is strictly less than c, though both nonzero. Then the residual network of that subgraph contains an edge from u to v with capacity $c - f$, as well as an edge in the opposite direction: from v to u with capacity f. This means that edges in the residual network exist in pairs: if there is an edge from a to b, there is one from b to a. Thus, for every vertex v in the fractional subgraph, in-degree(v) = out-degree(v). Therefore if the fractional subgraph contains edges (i.e. if there are any edges with fractional flow), it must contain a cycle. In fact, by a result from a previous problem set, every connected component with more than one vertex contains an Euler tour, which is certainly a cycle.

Next we justify the correctness of the algorithm's path-augmenting loop. We will show that each iteration of the loop decreases the number of edges with fractional flow. We assume the correctness of the DFS-based algorithm for finding cycles that was presented in recitation. When we have found a cycle in the residual network of the fractional subgraph, we can certainly iterate through it to find the edge e from u to v with minimum capacity C; i.e. every edge in the cycle has capacity greater than or equal to C. Then we can augment the original graph by increasing (or decreasing, depending on the direction of the edges in the residual network) the flow through each edge of the cycle by C. Since edge e in the residual network had capacity C, we are guaranteed to have saturated e in the residual network. This means that the edge between u and v in the original graph either has zero flow or is saturated. Since the maximum capacity of the edge is integral, we have thus ensured that the flow between u and v will now be integral, so at least one edge no longer has fractional flow. The algorithm certainly does not change any edge from integral flow to fractional flow, because it does not even consider, let alone change, any edge that is not on the subgraph of edges with fractional flow. Therefore the number of edges with fractional flow is strictly decreasing with each iteration of the loop, so it will eventually reach zero, at which time the algorithm will terminate. Finally, we note that this process still generates a valid max-flow. It is certainly valid to augment the path by C along the cycle, because we chose C such that each edge on the residual network had capacity

at least C. Notice that increasing every edge along a cycle by the same amount increases the flow into each vertex and the flow out of each vertex by the same amount. Thus, the flow continues to satisfy the flow conservation constraint. Moreover, the flow out of the source and into the sink is maintained invariant. Thus the algorithm correctly transforms the given fractional max-flow into an integral flow with the same value. This is the correct result.

Runtime analysis: The algorithm first needs to generate the fractional subgraph and its residual network. We showed above that this requires $O(V + E)$ time. Next we consider the work that is done during each iteration of the loop. We need to find a cycle by using DFS. Normally, DFS has runtime $O(V + E)$, but we can reduce this to $O(V)$ because we only need to continue searching until we find a cycle; there are only V vertices, so once DFS has considered V edges, it must have reached each vertex at least once, and thus we have found a cycle. The resulting cycle thus has $O(V)$ edges, so finding the minimum capacity of the edges takes $O(V)$ time, and incrementing the edges also takes $O(V)$ time. Thus each iteration of the loop requires $O(V)$ time. The loop cannot run more than E times, because each iteration of the loop reduces the number of edges with fractional flow by at least one, and there cannot be more than E edges with fractional flow to begin with. Thus the overall runtime of the algorithm is $O(VE)$.

6.046	Dan Ports
Problem 7-2	
No collab	Recitation #8

10/10

Algorithm: We will use essentially the same sweep-line approach for solving the problem as the output-sensitive algorithm in the lecture: we will sweep through the plane from left to right, keeping track of which horizontal segments currently intersect the sweep line, and whenever we hit a vertical segment we will count the number of horizontal segments it intersects. However, to maintain this information, we will make use of an order-statistic tree (CLRS chapter 14), sorted by y-coordinate. This data structure allows us to insert an element, remove an element, find an element by value, or find the rank of an element, each in $O(\lg n)$ time.

As we sweep from left to right, we need to consider $\Theta(n)$ "points of interest", defined as the left and right x-coordinates of each horizontal segment and the x-coordinate of each vertical segment. Using a sorting algorithm, we can place these in increasing order by x-coordinate in $O(n \lg n)$ time, along with pointers so we can find the other endpoints of the segment. We initialize a counter for the total number of intersection points to be zero. Then we consider each point of interest in increasing order by x-coordinate. If that point is the left endpoint of a horizontal segment, we find the y-coordinate and add it to our order-statistic tree, in $O(\lg n)$ time. If it is the right endpoint of a horizontal segment, we find the segment's y-coordinate and remove it from the order-statistic tree. If it is a vertical segment, we need to find the number of points which intersect it. To do this, we find the y-coordinates of the vertical segment — call them y_1 and y_2, and assume without loss of generality that $y_1 < y_2$. We find the element in the tree which has the smallest y-value greater than or equal to y_1 (call it v_1) and the element which has the largest y-value less than or equal to y_2 (call it v_2). Then the number of segments intersected by the vertical segment is $\text{RANK}(v_2) - \text{RANK}(v_1) + 1$. We increment the counter by this amount. Once we have reached the last point of interest, we return the value in the counter.

Runtime analysis: We can do all the preprocessing and sorting of the input points in $O(n \lg n)$ time, generating a list of points of interest in increasing order. There are 2 points of interest per horizontal segment and 1 per vertical segment, so $\Theta(n)$ total points of interest. For each point of interest, we require $O(\lg n)$ time: to add or remove an entry from the order-statistic tree if it is an endpoint of a horizontal segment, or to locate v_1 and v_2 and calculate their rank if it is a vertical segment. Thus the overall time requirement of the algorithm is $O(n \lg n)$.

Proof of correctness: We assume the correctness of the algorithm presented in the lecture. Since the bulk of this algorithm is essentially the same as that algorithm, we need only prove that maintaining the order-statistic tree gives the correct results. We use the invariant that the order-statistic tree contains the y-coordinate of each horizontal segment that the sweep line intersects. The procedures we follow when advancing the sweep line to the left or right endpoints of a horizontal line clearly maintain this invariant. Thus, when the sweep line is advanced to a vertical segment, it will intersect every horizontal line whose y-coordinate is between the endpoints of the vertical line. The number of such intersections is $\text{RANK}(v_2) - \text{RANK}(v_1) + 1$ for v_1 and v_2 as defined above (we add 1 because we want to include the points v_1 and v_2 in the count). The justification from this statement follows from the correctness of CLRS's order-statistic tree algorithm, which we assume.

10/10

6.046 **Dan Ports**

Problem 7-3

No collab Recitation #8

Algorithm: For this algorithm, we will also use a sweep-line approach. Essentially, as we sweep the line from left to right, we will keep track of the y-coordinates of all the "active" rectangles, and multiply the largest y-coordinate by the change in x-coordinate to determine the area to be added.

Specifically, given our input arrays $x_1[1 \ldots n]$, $x_2[1 \ldots n]$, and $y_1[1 \ldots n]$, we will begin by combining them into one array $A[1 \ldots n]$ containing all information for each rectangle, such that $A[i].\mathit{left} = x_1[i]$, $A[i].\mathit{right} = x_2[i]$, and $A[i].\mathit{top} = y[i]$. Assembling this array requires iterating through the other three arrays, in $\Theta(n)$ time. We next assemble an array $B[1 \ldots 2n]$ containing all x-coordinates of interest ($A[i].\mathit{left}$ and $A[i].\mathit{right}$ for every i) and a pointer to the associated element of array A. Assembling this array requires $\Theta(n)$ time, and we next sort it in increasing order by x-coordinate, requiring $\Theta(n \lg n)$ time.

Next we initialize a loop variable $i \leftarrow 1$, a counter $R \leftarrow 0$ in which we will accumulate our result, and a balanced BST T in which we will store the y-coordinates of all active rectangles. We iterate from $i = 1$ to $i = 2n$. On each iteration of the loop except the first ($i > 1$), we calculate the area required to paint the previous section of rectangles. This is calculated by taking the difference in x-coordinates between $B[i]$ and $B[i-1]$, and multiplying it by the maximum y-value contained in T (or zero if T is empty). We add that value to R. Then, if $B[i]$ points to the left coordinate of a rectangle, we follow the pointer and add the y-coordinate of that rectangle to T. Otherwise, if $B[i]$ points to the right coordinate of a rectangle, we follow the pointer and remove the y-coordinate of that rectangle from T. Once we have completed all $2n$ iterations of this loop, we return the result contained in R.

Runtime analysis: Generating the arrays A and B requires $\Theta(n)$ time, as shown above. Sorting B requires $\Theta(n \lg n)$ time. Each iteration of the loop requires finding the maximum element of T, and either adding an element to or removing an element from T, each of which can be done in $O(\lg n)$ time. The loop is run $2n = \Theta(n)$ times, so running the loop requires $O(n \lg n)$ time. The algorithm is thus overall $\Theta(n \lg n)$.

Proof of correctness: To prove correctness, we justify the claim that the total area required to paint the rectangles is the sum on i of the product between the distance between the points $B[i]$ and $B[i-1]$ and the maximum y-coordinate of a rectangle that starts at x-coordinate $B[i-1]$ or to the left and at x-coordinate $B[i]$ or to the right. Given a shape of overlapping rectangles, we can certainly divide it into many different rectangles without changing the area by splitting it at each of the "points of interest" listed in the array B. Then we have a number of rectangles, each of which has its left edge at the x-coordinate $B[i-1]$ and its right edge at $B[i]$ for some i. Thus for any fixed i, we can find a set E of rectangles which have left edge at $B[i-1]$ and right edge at $B[i]$. There must be some rectangle E_{max} which has the largest y-coordinate. Then every other rectangle in E is contained within E_{max} and can effectively be discarded. Applying this for every i, we see that the algorithm's approach of summing the product of the length of the interval $(B[i-1], B[i])$ and the largest y-coordinate of a rectangle that contains this interval is valid, giving the correct total area.

Finally, we note that it is easy to show that the algorithm correctly determines the aforementioned sum of products. This can be done by using an invariant: the tree T contains the y-coordinates of all rectangles that start at or before the x-coordinate $B[i-1]$ and end at $B[i]$ or after. Clearly the algorithm maintains this invariant by adding and deleting points in the tree whenever it encounters the left or right sides of a rectangle. Using this invariant, we know that taking the maximum y-coordinate contained in T and multiplying by the length of the interval gives the correct amount of paint required for the interval $(B[i-1], B[i])$. Thus the sum on i of this quantity is the total amount of paint required, and the algorithm gives the correct result.

10/10

6.046		Dan Ports
	Problem 7-4	
No collab		Recitation #8

Algorithm: This algorithm is closely based on the close-pair algorithm from lecture. We will divide the space into a grid where each side has length $\sqrt{2}$, and hash each grid space into a bucket. Specifically, we maintain an array of k buckets ($k = \Theta(n)$), and use a universal hash function h that maps $(\mathbb{R} \times \mathbb{R}) \mapsto \{1 \ldots k\}$. Then for every point $p_i = (x_i, y_i)$, we can compute its hash value $h(\lfloor \frac{x_i}{\sqrt{2}} \rfloor, \lfloor \frac{y_i}{\sqrt{2}} \rfloor)$. We iterate through the array of students, with i from 1 to n. For each i, we compute the hash value for the student's location $p_i = (x_i, y_i)$, and add him to the appropriate hash bucket. If the bucket contains 6 or more students, then we know that there must be one sociable student in the heavy bucket, unless it is a spurious hit caused by a hash collision. To ensure that this is not the case, we verify that there is a sociable student: for each student in the bucket, we check the distance to every other student in the bucket; if there exists some student such that the distance to at least five other students in the bucket is less than two meters, we have found a sociable student and we terminate immediately. Otherwise we continue adding students to buckets.

If we have placed every student in a hash bucket without finding a sociable student, then we iterate through the set of students, and for each student we check the distance from him to every student in his hash bucket and the hash buckets associated with the 24 grid cells that are 1 or 2 cells away from the student. If we find some student who is less than two meters away from at least five other students, then he is sociable and we terminate immediately. Otherwise we continue. If we have finished performing this check for each student, then there is no sociable student and we return false.

Proof of correctness: To justify the correctness of the algorithm, we first prove that if six students are in the same grid cell (which is equivalent to having six students in the same hash bucket, except in the case of a collision), there must be some sociable student. To see this, note that the grid sides have length $\sqrt{2}$. The furthest any two points can be away from each other is in the case where they are located at opposite corners diagonally, and in this case they are $\sqrt{2}\sqrt{2} = 2$ meters away from each other. Thus, each of the six students in the bucket are within two meters of each other. So a student in the bucket must be within two meters of five other students, meaning he is sociable.

Next we show that we need only consider the 25 neighboring buckets when searching for sociable students after filling the buckets. Any student who is not in one of these buckets must be at least $2\sqrt{2}$ meters away, or they would be in one of the neighboring grid cells. Certainly $2\sqrt{2} > 2$, so these students are more than two meters away, and we do not need to check them. Thus we need only check the students who are in the 25 hash buckets that are two grid cells or less away. (Actually, the 4 most distant of these 25 grid cells do not need to be considered, but this simplifies the description of the algorithm and does not affect the asymptotic runtime.)

Runtime analysis: This algorithm uses hashing, so it is randomized, and we will consider the expected runtime. Assume that computing the hash value requires constant time. Then, if we implement the buckets as a linked list, adding one student to the correct hash bucket requires constant time. If we keep track of the number of entries in the hash bucket, we can also check in constant time whether the bucket contains six students. Thus adding n students to the appropriate hash buckets requires $O(n)$ time. However, if we ever find six students in the same hash bucket,

we will need to verify that they are indeed sociable rather than being a spurious hit. This could potentially require many comparisons. However, the probability of a hash collision is low (the expected number is $\frac{n}{k}$), Thus, the expected runtime for this phase is still $O(n)$.

Next we show that scanning neighboring cells also requires $O(n)$ time. For each student, we want to scan 25 hash buckets. Assuming there are no collisions, each hash bucket will have no more than 5 students in it. Thus the total number of distances to be computed per student is less than 125, which is still constant. Thus applying this process of scanning neighboring cells for each of the n students requires $O(n)$ time. It is of course possible that we will have hash collisions, which can increase the runtime of this step. However, again we note that the probability of a hash collision is low, so the expected number of students in each bucket is still $O(1)$, so the expected runtime is still $O(n)$. Thus the algorithm has overall expected $O(n)$ runtime.