Chapter 33

Michelle Bodnar, Andrew Lohr

April 12, 2016

Exercise 33.1-1

Suppose first that the cross product is positive. We shall consider the angles that both of the vectors make with the positive x axis. This is given by $tan^{-1}(y/x)$ for both. Since the cross product is positive, we have that $0 < x_1y_2 - x_2y_1$, which means that $\frac{y_1}{x_1} < \frac{y_2}{x_2}$. however, since arctan is a monotone function, this means that the angle that p_2 makes with the positive x axis is greater than the angle that p_1 does, which means that you need to move in a clockwise direction to get from p_2 to p_1.

If the cross product is negative, this means that we have $\frac{y_1}{x_1} > \frac{y_2}{x_2}$ which means that the angle that p_1 makes is greater, which means that we need to go counter clockwise from p_2 to get to p_1.

Exercise 33.1-2

If the segment $\overline{p_ip_j}$ is vertical, then p_k could be colinear with p_i and p_j, but lie directly below them. Then $x_i = x_j = x_k$, so if we don't also check the y values we won't catch that p_k is not on the segment.

Exercise 33.1-3

The beauty of the fact that our sorting algorithms from earlier in the book were only comparison based is that if we can implement a comparison operation between any two elements that operates in constant time, then, we can use the earlier comparison based sorting algorithms as a black box to work on our data.

So, what we need to do is, given two indices i and j, decide whether the polar angle of p_i with respect to p_0 is larger or smaller than the polar angle of p_j with respect to p_0. This can be done with a single cross product. That is, we look at the cross product, $(p_1 - p_0) \times (p_2 - p_0)$. This is positive if we need to turn left from p_1 to get to p_2. That is, if it is positive, then the polar angle is greater for p_2 than from p_1. We similarly know that we are in the reverse situation if we have that this cross product is negative. The only tricky thing is that we could have two distinct elements p_i and p_j so that the cross product is still zero. The problem statement is unclear how to resolve these sorts of ties, because they have the same polar angle. We could just pick some arbitrary property of the points to resolve ties, such as we pick the point that is farther away from p_0 to be larger. Since we have a total ordering on the points that can be queried in constant time, we can use it in our $O(n \lg(n))$ algorithms from earlier on in the book.

Exercise 33.1-4

By Exercise 33.1-3 we can sort n points according to their polar angles with respect to a given point in $O(n \lg n)$ time. If points p_i, p_j, and p_k are colinear, then at two one of the following is true: (1) p_j and p_k have the same polar angle with respect to p_i, (2) p_i and p_k have the same polar angle with respect to p_j, or (3) p_i and p_j have the same polar angle with respect to p_k. Thus, it will suffice to do as follows: For each point p, compute the polar angle of all other points with respect to p. If there are any duplicates, those points are colinear. Since we must do this for each point, the algorithm has runtime $O(n^2 \lg n)$.

Exercise 33.1-5

Because, as stated in this definition of convex polynomials, we cannot have a vertex of a convex polygon be a convex combination of any two points of the boundary of the polynomial. This means that as we enter a particular vertex, we cannot have that it is colinear with the next vertex. Professor Amundsen's algorithm just rejects if both left and right turns are made. However, it should also reject if there is ever any vertex where no turn is made, because that vertex would then be a convex combination of the next and previous vertices.

Exercise 33.1-6

It will suffice to check whether or not the line segments $\overline{p_0p_3}$ and $\overline{p_1p_2}$ intersect, where $p_3 = (\max(x_1, x_2), y_0)$. We can do this in $O(1)$.

Exercise 33.1-7

Starting from the point p_0, pick an arbitrary direction, and consider the ray coming out in that direction. Instead of just counting the intersections with the sides of the polygon, we'll also count all the vertices that the ray intersects. for each side that it intersects, if it intersects the vertices at both sides, then we don't count that edge, because that means that the ray passes along that side. Lastly, if the ray passes through any vertex where both sides touching that vertex aren't of the previous type, we flip the parity of the count. Lastly, we say it is inside if the final count is odd. See the algorithm DETERMINE-INSIDE(P,p).

Exercise 33.1-8

Without loss of generality, assume that the interior of the polygon is to the right of the first segment $\overline{p_0p_1}$. We will examine segments of the polygon one at a

Algorithm 1 DETERMINE-INSIDE(P,p), P is a polygon, and p is a point

```
Let $S$ be the set of sides that the right horizontal ray intersects
Let $T$ be the set of vertices that the right horizontal ray intersects
Let $U$ be an empty set of sides
count = 0
for $s \in S$ do
    let $p_1$ and $p_2$ be the vertices at either side of s
    if $p_1 \in T$ and $p_2 \in T$ then
        put $s$ in $U$
    end if
    count++
end for
for $x \in T$ do
    let $y$ and $z$ be the sides that $x$ is touching
    if $y \in U$ or $z \in U$ then
        count++
    end if
end for
if count is odd then
    return inside
else
    return outside
end if
```

time. At any point, if we are at segment $\overline{p_ip_{i+1}}$ and if the next segment $\overline{p_{i+1}p_{i+2}}$ of the polygon turns right then, then we can compute the are of $\triangle p_ip_{i+1}p_{i+2}$, and reduce the problem to that of finding the area of the polygon without p_{i+1}, adding the area just computed. On the other hand, if we turn left then we need to compute the area of the polygon without p_{i+1}, but we need to subtract the area of $\triangle p_ip_{i+1}p_{i+2}$. Since we can compute the area of a triangle given its vertices in constant time, the runtime satisfies $T(n) = T(n-1) + O(1)$, so $T(n) = O(n)$.

Exercise 33.2-1

Suppose you split your set of lines into two equal sets, each of size $n/2$. Then, we will make half of them horizontal and close together, each above the next. That is, we'll put a horizontal line at $y = \frac{1}{k}$ for $k = 1, \ldots, n/2$ extending from -1 to 1. For the other half, we'll put lines along $x = \frac{1}{k}$ for $k = 1, \ldots, n/2$, extending from -1 to 1. Then, we'll have every line from the first set intersect every line from the second set. Therefore the total number of intersections is $\frac{n^2}{4}$, which is $\Theta(n^2)$.

Exercise 33.2-2

First suppose that a and b do not intersect. Let a_s, a_f, b_s, b_f denote the left and right endpoints of a and b respectively. Without loss of generality, let b have the leftmost left endpoint. If $\overline{b_sa_s}$ is to the right of $\overline{b_sb_f}$, then a is below b. Otherwise a is above b. Now suppose that a and b intersect. To decide which segment is on top, we need to determine whether the intersection occurs to the left or right of x. Assume that each point has x and y attributes. For example, $a_s = (a_s.x, a_s.y)$. The equation of the line through segment a is $y = m_1(x - a_s.x) + a_s.y$ where $m_1 = \frac{a_f.y - a_s.y}{a_f.x - a_s.x}$. The equation of the line through segment b is $y = m_2(x - b_s.x) + b_s.y$ where $m_2 = \frac{b_f.y - b_s.y}{b_f.x - b_s.x}$. Setting these equal to each other gives

$$x = \frac{b_s.y - m_2b_s.x - a_s.y + m_1a_s.x}{m_1 - m_2}.$$

Let $x = x_0$ be the equation of the sweep line at which we want to test the relationship between a and b. We need to determine whether or not $x < x_0$, but without using division. To do this, we'll need to clear denominators. $x < x_0$ is equivalent to

$$b_s.y - m_2b_s.x - a_s.y + m_1a_s.x < (m_1 - m_2)x_0$$

which is equivalent to this gross mess, which fortunately requires only addition, subtraction, multiplication, and comparison, so it is numerically stable:

$$(a_f.x - a_s.x)(b_f.x - b_s.x)(b_s.y - a_s.y) - (a_f.x - a_s.x)(b_f.y - b_s.y)b_s.x + (b_fx - b_s.x)(a_f.y - a_s.y)a_s.x$$
$$< (b_f.x - b_s.x)(a_f.y - a_s.y)x_0 - (a_f.x - a_s.x)(b_f.y - b_s.y)x_0.$$

Exercise 33.2-3

It looks like the moral of this book is that the only time that a professor can be right is when he's disagreeing with another professor. Professor Dixon is correct.

It will not necessarily print the leftmost intersection first. The intersection that it prints first will be the pair of lines such that both lines have their endpoints show up first in the lexicographical ordering on line 2. An example is, suppose we have the lines $\{\{(0,1000),(2,2000)\},\{(0,1001),(2,1001)\},\{(0,0),(1,2)\},\{(0,2),(1,0)\}\}$. Then, the first two lines have the leftmost intersection, but the intersection between the last two lines will be printed out first.

The procedure will not necessarily display all intersections, in particular, suppose that we have the line segments $\{\{(0,0),(4,0)\},\{(0,1),(4,-2)\},\{(0,2),(4,-2)\},\{(0,3),(4,-1)\}\}$. There are intersections of the first line segment with each of the other line segments at 1,2, and 3. However, we cannot detect the intersection at 2 because the line segment from $(0,2)$ to $(4,-2)$ is not adjacent to the horizontal line segment in the red-black tree either when we process left endpoints or right endpoints.

Exercise 33.2-4

An n vertex polygon $\langle p_0, p_1, \ldots, n_{n-1}\rangle$ is simple if and only if the only intersections of the segments $\overline{p_0p_1}, \overline{p_1p_2}, \ldots, \overline{p_{n-1}p_0}$ of the boundary are between consecutive segments $\overline{p_ip_{i+1}}$ and $\overline{p_{i+1}p_{i+2}}$ at the point p_{i+1}. We run the usual ANY-SEGMENTS-INTERSECT algorithm on the segments which make up the boundary of the polygon, with the modification that if an intersection is found, we first check if it is an acceptable one. If so, we ignore it and proceed. Since we can check this in $O(1)$, the runtime is the same as ANY-SEGMENTS-INTERSECT.

Exercise 33.2-5
Construct the set of line segments which correspond to all the sides of both polygons, then just use the algorithm from this section to see if any pair of them intersect. If we are in the fringe case that some segment is vertical, just rotate the whole picture by some epsilon. This won't change whether or not there is an intersection.

Exercise 33.2-6

We can use a modified version of the intersecting-segments algorithm to solve this problem. We'll first associate left and right endpoints to each disk. If disk D has radius r and center (x,y), define its left endpoint to be $(x-r,y)$ and its right endpoint to be $(x+r,y)$. Begin by ordering the endpoints of the disks first by left-right position. If two endpoints have the same x-coordinate, then covertical left endpoints come before right endpoints. Within these, order by y-coordinates from low to high. We'll use the same event point schedule as for the intersecting segments problem. Maintain a sweep-line status that gives the relative order of the segments of the disks, where the segment associated to each disk is the segment formed by its left and right endpoints. When we encounter a left endpoint, we add the associated disk to the sweep-line status. When we encounter a right endpoint, we delete the disk from the sweep-line status. Consider the first time two disks become consecutive in the ordering. Let their centers be (x_1, y_1) and (x_2, y_2), and their radii be r_1 and r_2. Check if $(x_2 - x_1)^2 + (y_2 - y_1)^2 \leq (r_1 + r_2)^2$. If yes, then the two circles intersect. Otherwise they don't. Since we can check this in $O(1)$, and there are only $2n$ points which are added, we make at most $4n$ checks in total. Sorting the points takes $O(n \lg n)$, so the total runtime is $O(n \lg n) + O(n) = O(n \lg n)$.

Exercise 33.2-7

We preform a slight modification to what Professor Mason suggested in exercise 33.2-3. Once we have found an intersection, we then keep considering elements further and further away in the red black tree until we no longer have an intersection. Since all the tree operations only take time $O(\lg(n))$, and we are only doing an additional one on top of the original algorithm for each of the intersections that we found, we have that the additional runtime is $O(k \lg(n))$ so, the total runtime is $O((n + k) \lg(n))$.

Exercise 33.2-8

Suppose that at least 3 segments intersect at the same point. It is clear that if ANY-SEGMENTS-INTERSECT returns true, then it must be correct. Now we will show that it must return true if there is an intersection, even if it occurs as an intersection of 3 or more segments. Suppose that there is at least one intersection, and that p is the leftmost intersection point, breaking ties by choosing the point with the lowest y-coordinate. Let $a_1, a_2, \ldots, a_k$ be the segments that intersect at p. Since no intersections occur to the left of p, the order given by T is correct at all points to the left of p. Let z be the first sweep line at which some pair a_i, a_j is consecutive in T. Then z must occur at or to the left of p. Let i be the smallest number such that there exists a j such that a_i and a_j are consecutive in T, and assume that we choose the smallest j possible, and let q be the event point at which a_i and a_j become consecutive in the total preorder. If p is on z, then we must have $q = p$, and at this point the intersection is detected. As in the proof of correctness given in the section, the ordering of our endpoints allows us to detect this even if p is the left endpoint of a_i and the right endpoint of a_j. If p is not on z then q is to the left of p, and when we process q no other intersections have occurred, so the ordering in T is correct, and the algorithm correctly identifies the intersection between a_i and a_j.

Exercise 33.2-9

In the original statement of the problem, we are putting points with lower y-coordinates first. This means that when we are processing our vertical segment, we want its lower bound to of already been processed by the time we process any of the left endpoints of other lines that may intersect the given line. Also, we don't want to remove the segment until we have already processed all the right endpoints of the lines that may of intersected it, which means we want it's upper bound to be dealt with in the second pass (the right endpoint pass). Again, since we process lower y-values first, this means that we have it added to our tree before we process anything it could intersect and have it removed after processing everything it could intersect.

If one or both of the segments are vertical at x in exercise 33.2-2, then testing whether they intersect is just a matter of looking to see if the other line is less than the upper bound and more than the lower bound at the given x value. otherwise we just see if it's more than the upper bound or less than the lower bound to see which direction the inequality should go.

Exercise 33.3-1

To see this, note that p_1 and p_m are the points with the lowest and highest polar angle with respect to p_0. By symmetry, we may just show it for p_1 and we would also have it for p_m just by reflecting the set of points across a vertical line. To a contradiction, suppose we have the convex hull doesn't contain p_1. Then, let p be the point in the convex hull that has the lowest polar angle with respect to p_0. If p is on the line from p_0 to p_1, we could replace it with p_1 and have a convex hull, meaning we didn't start with a convex hull. If we have that it is not on that line, then there is no way that the convex hull given contains p_1, also contradicting the fact that we had selected a convex hull.

Exercise 33.3-2

Let our n numbers be $a_1, a_2, \ldots, a_n$ and f be a strictly convex function, such as e^x. Let $p_i = (a_i, f(a_i))$. Compute the convex hull of $p_1, p_2, \ldots, p_n$. Then every point is in the convex hull. We can recover the numbers themselves by looking at the x-coordinates of the points in the order returned by the convex-hull algorithm, which will necessarily be a cyclic shift of the numbers in increasing order, so we can recover the proper order in linear time. In an algorithm such as GRAHAM-SCAN which starts with the point with minimum y-coordinate, the order returned actually gives the numbers in increasing order.

Exercise 33.3-3

Suppose that p and q are the two furthest apart points. Also, to a contradiction, suppose, without loss of generality that p is on the interior of the convex hull. Then, construct the circle whose center is q and which has p on the circle. Then, if we have that there are any vertices of the convex hull that are outside this circle, we could pick that vertex and q, they would have a higher distance than between p and q. So, we know that all of the vertices of the convex hull lie inside the circle. This means that the sides of the convex hull consist of line segments that are contained within the circle. So, the only way that they could contain p, a point on the circle is if it was a vertex, but we suppsed that p wasn't a vertex of the convex hull, giving us our contradiction.

Exercise 33.3-4

We simply run GRAHAM-SCAN but without sorting the points, so the runtime becomes $O(n)$. To prove this, we'll prove the following loop invariant: At the start of each iteration of the for loop of lines 7-10, stack S consists of, from bottom to top, exactly the vertices of $\text{CH}(Q_{i-1})$. The proof is quite similar to the proof of correctness. The invariant holds the first time we execute line 7 for the same reasons outline in the section. At the start of the i^{th} iteration, S contains $\text{CH}(Q_{i-1})$. Let p_j be the top point on S after executing the while loop of lines 8-9, but before p_i is pushed, and let p_k be the point just below p_j on S. At this point, S contains $\text{CH}(Q_j)$ in counterclockwise order from bottom to top. Thus, when we push p_i, S contains exactly the vertices of $\text{CH}(Q_j \cup \{p_i\})$.

We now show that this is the same set of points as $\text{CH}(Q_i)$. Let p_t be any point that was popped from S during iteration i and p_r be the point just below p_t on stack S at the time p_t was popped. Let p be a point in the kernel of P. Since the angle $\angle p_r p_t p_i$ makes a nonelft turn and P is star shaped, p_t must be in the interior or on the boundary of the triangle formed by p_r, p_i, and p. Thus, p_t is not in the convex hull of Q_i, so we have $\text{CH}(Q_i - \{p_t\}) = \text{CH}(Q_i)$. Applying this equality repeatedly for each point removed from S in the while loop of lines 8-9, we have $\text{CH}(Q_j \cup \{p_i\}) = \text{CH}(Q_i)$.

When the loop terminates, the loop invariant implies that S consists of exactly the vertices of $CH(Q_m)$ in counterclockwise order, proving correctness.

Exercise 33.3-5

Suppose that we have a convex hull computed from the previous stage $\{q_0, q_1, \dots, q_m\}$, and we want to add a new vertex, p in and keep track of how we should change the convex hull. First, process the vertices in a clockwise manner, and look for the first time that we would have to make a non-left to get to p. This tells us where to start cutting vertices out of the convex hull. To find out the upper bound on the vertices that we need to cut out, turn around, start processing vertices in a clockwise manner and see the first time that we would need to make a non-right. Then, we just remove the vertices that are in this set of vertices and replace the with p. There is one last case to consider, which is when we end up passing ourselves when we do our clockwise sweep. Then we just remove no vertices and add p in in between the two vertices that we had found in the two sweeps. Since for each vertex we add we are only considering each point in the previous step's convex hull twice, the runtime is $O(nh) = O(n^2)$ where h is the number of points in the convex hull.

Exercise 33.3-6

Algorithm 2 ONLINE-CONVEX-HULL

```
let $P = \{p_0, p_1, \dots p_m\}$ be the convex hull so far listed in counterclockwise order.
let $p$ be the point we are adding
i=1
while going from $p_{i-1}$ to $p_i$ to p is a left turn and $i \neq 0$ do
    i++
end while
if i==0 then
    return P
end if
j=i
while going from $p_{i+1}$ to $p_i$ to $p$ is a right turn and $j \geq i$ do
    j–
end while
if $j < i$ then
    insert p between $p_j$ and $p_i$
else
    replace $p_i, \dots p_j$ with $p$.
end if
```

First sort the points from left to right by x coordinate in $O(n \lg n)$, breaking ties by sorting from lowest to highest. At the i^{th} step of the algorithm we'll compute $C_i = \text{CH}(\{p_1, \dots, p_i\})$ using C_{i-1}, the convex hull computed in the previous step. In particular, we know that the rightmost of the first i points will be in C_i. The point which comes before p_i in a clockwise ordering will be the first point q of C_{i-1} such that $\overline{qp_i}$ does not intersect the interior of C_{i-1}. The point which comes after p_i will be the last point q' in a clockwise ordering of the vertices of C_{i-1} such that $\overline{p_i q'}$ does not intersect the interior of C_{i-1}. We can find each of these points in $O(\lg n)$ using a binary search. Here, assume that q and q' are given as the positions of the points in the clockwise ordering). This follows because we're searching a set of points which already forms a convex hull, so the segments $\overline{p_j p_i}$ will intersect the interior of C_{i-1} for the first k_1 points, not intersect for the next k_2 points, and intersect for the last k_3 points, where any of k_1, k_2, or k_3 could be 0. Once found, we can delete every point between q and q'. Since a point is deleted at most once and we store things in a red-black tree, the total runtime of all deletions is $O(n \lg n)$. Since we insert a total of n points, each taking $O(\lg n)$, the total runtime is thus $O(n \lg n)$. See the algorithm below:

Exercise 33.4-1

The flaw in his plan is pretty obvious, in particular, when we select line l, we may be unable perform an even split of the vertices. So, we don't neccesarily have that both the left set of points and right set of points have fallen to roughly half. For example, suppose that the points are all arranged on a vertical

Algorithm 3 INCREMENTAL-METHOD($p_1, p_2, \dots, p_n$)

```
  if $n \leq 3$ then
      return $(p_1, \dots, p_n)$
  end if
  Use Merge Sort to sort the points by increasing $x$-coordinate, breaking ties by requiring increasing $y$-coordinate
  Initialize an red-black tree $C$ of size 3 with entries $p_1, p_2$, and $p_3$
  for $i = 4$ to $n$ do
      Let $q$ be the result of binary searching for the first point of $C_{i-1}$ such that $\overline{qp_i}$ doesn't intersect the interior of $C_{i-1}$
      Let $q'$ be the result of binary searching for the last point of $C_{i-1}$ such that $\overline{q'p_i}$ doesn't intersect the interior of $C_{i-1}$
      Delete $q+1, q+2, \dots, q'-1$ from $C$
      Insert $p_i$ into $C$
  end for
```

line, then, when we recurse on the the left set of points, we haven't reduced the problem size AT ALL, let alone by a factor of two. There is also the issue in this setup that you may end up asking about a set of size less than two when looking at the right set of points.

Exercise 33.4-2

Since we only care about the shortest distance, the distance δ' must be strictly less than δ. The picture in Figure 33.11(b) only illustrates the case of a nonstrict inequality. If we exclude the possibility of points whose x coordinate differs by exactly δ from l, then it is only possible to place at most 6 points in the $\delta \times 2\delta$ rectangle, so it suffices to check on the points in the 5 array positions following each point in the array Y'.

Exercise 33.4-3

In the analysis of the algorithm, most of it goes through just based on the triangle inequality. The only main point of difference is in looking at the number of points that can be fit into a $\delta \times 2\delta$ rectangle. In particular, we can cram in two more points than the eight shown into the rectangle by placing points at the centers of the two squares that the rectangle breaks into. This means that we need to consider points up to 9 away in Y' instead of 7 away. This has no impact on the asymptotics of the algorithm and it is the only correction to the algorithm that is needed if we switch from L_2 to L_1.

Exercise 33.4-4

We can simply run the divide and conquer algorithm described in the section, modifying the brute force search for $|P| \leq 3$ and the check against the next 7 points in Y' to use the L_∞ distance. Since the L_∞ distance between two points is always less than the euclidean distance, there can be at most 8 points in the $\delta \times 2\delta$ rectangle which we need to examine in order to determine whether the closest pair is in that box. Thus, the modified algorithm is still correct and has the same runtime.

Exercise 33.4-5

We select the line l so that it is roughly equal, and then, we won't run into any issue if we just pick an arbitrary subset of the vertices that are on the line to go to one side or the other. Since the analysis of the algorithm allowed for both elements from P_L and P_R to be on the line, we still have correctness if we do this. To determine what values of Y belong to which of the set can be made easier if we select our set going to P_L to be the lowest however many points are needed, and the P_R to be the higher points. Then, just knowing the index of Y that we are looking at, we know whether that point belonged to P_L or to P_R.

Exercise 33.4-6

In addition to returning the distance of the closest pair, the modify the algorithm to also return the points passed to it, sorted by y-coordinate, as Y. To do this, merge Y_L and Y_R returned by each of its recursive calls. If we are at the base case, when $n \leq 3$, simply use insertion sort to sort the elements by y-coordinate directly. Since each merge takes linear time, this doesn't affect the recursive equation for the runtime.

Problem 33-1

a. We need just iteratively apply Jarvis march. The first march takes time $O(n|CH(Q_1)|)$, the next time $O(nCH(Q_2))$, and so on. So, since each point in Q appears in exactly one convex hull, as we take off successive layers, we have

$$\sum_i O(n|CH(Q_i)|) = O(n\sum_i |CH(Q_i)|) = O(n^2)$$

b. Suppose that the elements $r_1, r_2, r_3, \ldots r_\ell$ are the points that we are asked to sort. We will construct an instance of the convex layers problem, whose solution will tell us what the sorted order of $\{r_i\}$ is. Since we can't comparison sort quickly, and this would provide a solution of sorting based on a convex layers algorithm, it would mean that we cannot find a convex layers algorithm that takes time less than $\Omega(n\lg(n))$.

   Suppose that all the $\{r_i\}$ are positive. If they aren't, we can in linear time find the one with the smallest value and subtract that value minus one from each of them. We will select our 4ℓ points to be

$$P = \{(r_i, 0)\} \cup \{(0, \pm i) | i = 1, 2, \dots \ell\} \cup \{(-i, 0) | i = 1, 2, \dots \ell\}$$

Note that all of the points in this set are on the coordinate axes. So, every layer will contain one point that lies on each of the four half axes coming out of the origin. Looking at the points that lie on the positive x axis, they will correspond to the original points that we wanted to sort. Also, by looking at the outermost layer and going inwards, we are reading off the points $\{r_i\}$ in order of decreasing value. Since we have only increased the size of the problem by a constant factor, we haven't changed the asymptotics. In particular, if we had some magic algorithm for convex layers that was $o(n \lg(n))$, we would then have an algorithm that was $o(n \lg(n))$.

See also the solution to 33.3-2

Problem 33-2

a. Suppose that $y_i \leq y_{i+1}$ for some i. Let p_i be the point associated to y_i. In layer i, p_i is the leftmost point, so the x-coordinate of every other point in layer i is greater than the x-coordinate of p_i. Moreover, no other point in layer i can have y coordinate greater than p_i, since that would imply it dominates p_i. Let q_{i+1} be the point of layer $i + 1$ with y-coordinate y_{i+1}. If q_{i+1} is to the left of p_i, then q_{i+1} cannot be dominated by any point in L_i since every point in L_i is to the right of and below p_i. Moreover, if q_{i+1} is to the right of p_i then q_{i+1} dominates p_i, which can't happen. Thus, q_{i+1} cannot be weakly to the left or right of p_i, a contradiction. Thus $y_i > y_{i+1}$.

b. First suppose $j \leq k$. Then for $1 \leq i \leq j - 1$ we have that (x, y) is dominated by the point in layer i with y-coordinate y_i, so (x, y) is not in any of these layers. Since (x, y) is the leftmost point and $y_j < y$, and all other points in layer j have lower y coordinate, no point in layer j dominates (x, y). Moreover, since it is leftmost, no other point can be dominated by (x, y). Thus, (x, y) is in L_j, as well as all other points previously in L_j. The other layers are unaffected since we no longer consider (x, y) when computing them. Thus the layers of Q' are identical to the maximal layers of Q, except that $L_j = L_j \cup (x, y)$.

   Now suppose $j = k + 1$. Then (x, y) is dominated by each point in layer i with y-coordinate y_i, so it can't be in any of the first k layers. This implies that it is in a layer of its own, $L_{k+1} = \{(x, y)\}$.

c. First sort the points by x coordinate, with the highest coordinate first. Process the points one at a time. For each point, find the layer in which it belongs as described in part b, creating a new layer if necessary. We can maintain lists of the layers in sorted order by y coordinate of the leftmost element of each list. In doing so, we can decide which list each new point belongs to in $O(\lg n)$ time. Since there are n points to process, the runtime after sorting is $O(n \lg n)$. The initial sorting takes $O(n \lg n)$, so the total runtime is $O(n \lg n)$.

d. We'll have to modify our approach to deal with points having the same x- or y-coordinate. In particular, if two points have the same x-coordinate then when we go to place the second one, the old algorithm would have us put it in the same layer as the first one. We'll compensate for this as follows. Suppose we wish to add the point (x, y). Let j be the minimum index such that $y_j < y$. If the x-coordinate of the leftmost point of L_j is equal to x, then we need to create a new list L' which lives between L_{j-1} and L_j. Using red-black trees we can update the information in $O(\lg n)$ time. Otherwise, we add (x, y) to L_j as usual. If $j = k + 1$, then create a new layer L_{k+1}. Two points having the same y-coordinate doesn't actually cause any difficulty because of the strict inequality required for the check described in part b.

Problem 33-3

a. Take a convex hull of the set of all the ghostbusters and the ghosts. If the convex hull doesn't consist of either all ghosts or all busters, we can just pick an edge of the convex hull that joins a buster and a ghost, Since all of the other points lie on the same side of that line, the number of ghosts and busters will be n-1 and so will be equal.

   So, assume that the convex hull does not contain one of both types. Since there is symmetry between ghosts and ghostbusters, suppose the convex hull is entirely made of ghostbusters. Pick an arbitrary ghostbuster on the convex hull, and that he's facing somewhere inside the convex hull. Have him/her initially pointing his proton pack just to the left the person furthest to his right and have him slowly start turning left. We know that initially there are more ghostbusters than ghosts to his right. We also know that by the time he is just to the right of the person furthest to his left there are more ghosts to his right than ghostbusters. This means at some point he must of gone from having more ghostbusters to his right to having more ghosts to his right. In order to have this happen he had to of just passed a ghost. So, he is then paired up with that ghost.

b. We just keep iterating the first part of this procedure, applying it separately to all the ghosts and ghostbusters to each of the sides of the line. We have that no beam will cross because the beams for each stays entirely on that side of the line. This gives us, for some $n \le k > 0$, the recurrence

$$T(n) = T(n - k) + T(k - 1) + n \lg(n)$$

   This has the worst case when either k is really tiny or really close to n. Therefore, the worst case solution to this recurrence is $O(n^2 \lg(n))$.

Problem 33-4

a. Let a be given by endpoints (a_x, a_y, a_z) and (a'_x, a'_y, a'_z) and b be given by endpoints (b_x, b_y, b_z) and (b'_x, b'_y, b'_z). Compute, using cross products, whether or not segments $\overline{(a_x, a_y)(a'_x a'_y)}$ and $\overline{(b_x, b_y)(b'_x, b'_y)}$ intersect in constant time, as described earlier in the chapter. If they do, then either a or b is above the other one. If not, then they are unrelated. If they are related, we need to determine which of a and b are on top. In this case, there exist λ_1 and λ_2 such that

$$a_x + \lambda_1(a'_x - a_x) = b_x + \lambda_2(b'_x - b_x)$$

and

$$a_y + \lambda_1(a'_y - a_y) = b_y + \lambda_2(b'_y - b_y).$$

In other words, we get intersection when we project to the xy-plane. We can solve for λ_1 and λ_2. This requires division at first blush, but we shall see in a moment that this isn't necessary. In particular, a is above b if and only if $a_z + \lambda_1(a'_z - a_z) \geq b_z + \lambda_1(b'_z - b_z)$. By multiplying both sides by $(a'_x - a_x)(b'_y - b_y - (a'_y - a_y)(b'_x - b_x))$ we clear all denominators, so we need only perform addition, subtraction, multiplication, and comparison to determine whether a is on top. Moreover, we can do this in constant time.

b. Make a graph whose vertices are each of the n points. Find each pair of overlapping sticks. If a is above b, then draw a directed edge from a to b. Then perform a topological sort to determine an ordering of picking up the sticks. If such an ordering exists, then we use it. Otherwise there is no legal way to pick up the sticks. Since there could be as many as $O(n^2)$ instances of a point a being above a point b, there could be $\Theta(n^2)$ edges in the graph, so the runtime is $O(n^2)$.

Problem 33-5

a. Pick one point on one of the convex hulls, and look at the point on the other that has the lowest polar angle. Then, start marching counter clockwise around the first hull until it would require a non-right turn to go to the point selected before. Do the same thing, picking a point and looking at the point on the second polygon with highest polar angle, and keep marching in a clockwise direction until getting to the particular point would require a non-right. Cut out all the vertices between these two places we stopped inclusive. In their place put the vertices of the other convex polygon that are between to two selected vertices of it, inclusive.

b. Let P_1 be the first $\lceil n/2 \rceil$ points, and let P_2 be the second $\lfloor n/2 \rfloor$ points. Since the original set of points were selected independently from the sparse distribution, both the sets P_1 and P_2 were selected from a sparse distribution. This means that we have that $|CH(P_1)| \in O(n^{1-\epsilon})$ and also, $|CH(P_2)| \in O(n^{1-\epsilon})$. Then, by applying the procedure from part a, we have the recurrence $T(n) \leq 2T(n/2) + |CH(P_1)| + |CH(P_2)| = 2T(n/2) + O(n^{1-\epsilon})$.

By applying the master theorem, we see that this recurrence has solution $T(n) \in O(n)$.