Advanced Algorithms, Fall 2011

Homework #8: Solutions

1. Devise and analyze (in worst-case terms) a divide-and-conquer algorithm for the following problem. Given a set of rectangles, all bases of which lie on the x-axis, determine the upper envelope of the collection of rectangles.

   Would your algorithm still work if the set contained triangles rather than rectangles (still with their base on the x-axis, of course)? How far can you generalize the types of shapes for which your algorithm will work?

   This problem is also known as the "Manhattan skyline" problem—every rectangle is a building and the upper envelope is what is solid and hides the sky. The skyline can be viewed as an ordered list of horizontal segments, each beginning at the abscissa at which the preceding one ended; this viewpoint leads to a representation as an alternating sequence of abscissae and ordinates, where each ordinate indicates the height of its corresponding segment:

   $$(x_1, y_1, x_2, y_2, \ldots, x_{n-1}, y_{n-1}, x_n) \text{ with } x_1 < x_2 < \cdots < x_n.$$

   By convention, we set the skyline at height zero until x_1 is reached and back to zero after x_n is passed. We use the same representation for a single rectangle (a building), namely (x_1, y_1, x_2), which specifies the rectangle with vertex coordinates $(x_1, 0)$, (x_1, y_1), (x_2, y_1), and $(x_2, 0)$.

   The main decision to make is whether to use a geometric D&C or a set-based one. For the algorithms we saw in class, the set-based D&C could run faster under the right circumstances, so we go with that here as well. Thus we need a merging procedure that merges two arbitrary skylines—and does so as efficiently as possible. Note that the abscissae of the merged skyline form a subset of the abscissae of the two skylines, since every change in the skyline corresponds to either the left or right edge of a rectangle. Thus merging skylines, like intersecting convex polygons, is output-sensitive. The merging step runs in $\Theta(n_1 + n_2)$ time, where n_1 and n_2 are the sizes of the two skylines and returns a skyline of size varying from 3 to $n_1 + n_2 + 1$. The paradigm for the merging step remains that of mergesort, but it must be modified to accommodate the output-sensitivity, as well as to account for the special cases introduced by the geometric nature of the problem. The worst-case running time of the entire algorithm is thus given by

   $$\begin{cases} T(n) = 2T(n/2) + \Theta(n) \\ T(1) = \Theta(1). \end{cases}$$

   Thus the algorithm runs in $\Theta(n \log n)$ time in the worst case. With random or "average" data, the algorithm may run considerably faster because of its output-sensitivity.

   It is quite possible to get a $\Theta(n \log n)$ solution by using geometric divide-and-conquer, but then we have to sort before we can divide and so lose any output-sensitivity.

Now, can we extend this approach to other shapes? The representation will have to change to accommodate new shapes, but the main issue is optimality, for results or for running time. With triangles instead of rectangles, things obviously work and the only change is that successive numbers are now tops and bottoms of a sawtooth skyline. If we replace the rectangles by any convex shape where the bottom edge lies on the x axis and spans the entire range of abscissae of vertices of the piece, everything still works, although now we get a sawtooth with upwards convex pieces. On the other hand, if the pieces have vertices outside the range defined by the bottom edge (some buildings have overhangs), or the pieces are not convex, then things fall apart—the same algorithm fails to return the correct skyline in all cases and a correct algorithm gets very expensive, because the skyline can get arbitrarily complicated.

2. Recall that Fibonacci numbers are defined by the recurrence $F(n) = F(n-1) + F(n-2)$, with initial conditions $F(0) = 0$ and $F(1) = 1$. Computing the nth Fibonacci number is easily done in $n-1$ additions. However, we can use divide-and-conquer techniques to compute the nth Fibonacci number with $\Theta(\log n)$ arithmetic operations. Derive such an algorithm.

   In order to compute something that large with just $O(\log n)$ operations, we will need to use multiplication—using just additions must take linear time, since the defining recurrence uses additions as efficiently as possible. Thus we need a new recurrence that gives us $F(n)$ as a function of much lower indices, such as $\frac{n}{2}$. Let us start with $F(2n+1)$, then tackle $F(2n)$; we can write

   $$F(2n+1) = F^2(n+1) + F^2(n)$$

   (This is easily verified by induction.) Similarly, we can write

   $$F(2n) = F^2(n) + 2 \cdot F(n) \cdot F(n-1)$$

   (This is also easily verified by induction.) Together, the two give us a way to compute $F(m)$ and $F(m-1)$ by computing $F(m/2)$ and its predecessor, and thus to run with a logarithmic number of operations.

3. Convexity is a very strong property, but a weaker version often suffices. Consider this version: a polygon is “locally convex” if there exists some point p in the interior of the polygon such that, for each vertex v of the polygon, the line segment $\overline{pv}$ lies entirely within the polygon. (This is much as in the geometric definition of a convex polygon, except that only one endpoint of the segment is arbitrary.)

   Design and analyze a randomized algorithm that, given just the polygon (by a counterclockwise list of its vertices along the perimeter), decides in linear expected time whether the polygon is locally convex.

   Polygons that are locally convex are also known as “star” polygons, since every point in the interior is on a ray originating in the special point. The crucial observation to make is the following: for such a polygon with special point p, that point p lies in the intersection of the halfplanes defined by successive edges of the polygon. And the crucial theorem is that this relationship is an “if and only if:” a polygon is locally convex *iff* the intersection of the halfplanes determined by its edges (using left sides and a counterclockwise traversal of the perimeter) is nonempty. So, instead of testing for the existence of p, we will test whether the intersection of these halfplanes is empty, using randomization.

We use a randomized incremental approach: start with the two consecutive edges (halfplanes) that define the vertex of the polygon with the largest ordinate. We will maintain the highest vertex, v^*, of the intersection polygon; we have just initialized it to be the highest vertex of the input polygon. Now repeatedly add the next, randomly selected, halfplane and update the selection of the highest vertex. If the next halfplane includes v^*, then that halfplane is redundant for the intersection figure and discarded. Otherwise, we must update v^*. Note that the new v^* must lie on the new halfplane's boundary; in fact, it is simply the higher of the two intersection points of that halfplane's boundary with the current convex intersection (if the halfplane boundary happens to be horizontal, then any point along the line segment works, but we can still choose one of the two endpoints). We can find these intersection points by brute force, by simply intersecting each of the halfplanes already processed with the new boundary line. This takes time linear in the number of halfplanes already processed, so the worst-case running time is clearly quadratic, because each successive halfplane added to the collection could modify v^*. However, since we randomized the order of addition, we now want the probability that the next halfplane added will modify v^*. We do this as we handled the smallest enclosing disk problem, by assuming that the ith halfplane was just added and looking at the probability that this addition changed v^*.

Since we assume, as always, that all points are in general position, v^* is determined by the intersection of two boundary lines, from the i such lines (halfplanes) processed so far. Since we randomized the order in which the halfplanes are processed, the probability that the ith halfplane added is one of the two that define the current v^* is bounded by $\frac{2}{i}$, and thus the expected running time of our algorithm is

$$\sum_{i=1}^{n}(1-\frac{2}{i})\cdot\Theta(1)+\frac{2}{i}\Theta(i)$$

where the first contribution comes from halfplanes that contained the current v^* when added and the second comes from halfplanes that modified v^* and so caused linear work. This is the same sum we encountered with the smallest enclosing disk and is in $\Theta(n)$.

It is actually not hard to derandomize this algorithm and thus obtain a deterministic linear-time algorithm to test whether the intersection is empty. Simply go around the perimeter and compute on the fly the intersection of the half-planes defined by the edges seen so far. The trick is that intersecting the existing intersection polygon with the next half-plane can be done in *amortized* constant time, because we know a lot about the next half-plane. We maintain the tangency points to the current intersection from the next vertex on the perimeter of the given polygon and use them in handling the next edge; note that maintaining these tangency points is simply a matter of turning around the (convex) intersection figure and, in the worst case, will have required us to turn all the way around the intersection figure through the execution of the algorithm, which clearly takes linear time overall and thus constant amortized time per operation. With the help of the tangency points, it is a simple matter to compute the intersection with the new half-plane, again by walking the boundary of the current intersection polygon, and again for a total cost of O(n) overall.

4. You are in a crowd of n people; over half of them belong to the same secret party. You go around asking questions; the only question you can ask is "are these two people (pointing at one, then at another) from the same party?" Everyone in the crowd is truthful.

   Use a divide-and-conquer approach to identify with no possibility of error one person who belongs to the secret party. (Note that it is possible to solve this problem in linear time by other techniques; the divide-and-conquer algorithm will impose an extra factor of $\log n$, but it is what is asked of you here.)

   The linear-time algorithm is both clever and deceptively simple—but left to the reader to design... The divide-and-conquer algorithm is just simple!

   To simplify the reasoning and the analysis, assume that n is a power of two, $n = 2^k$ for some $k > 0$. Then, if over half of these n people belong to the secret party, at least $2^{k-1} + 1$ people belong to this party and thus, in any partition of the n people into two subsets of 2^{k-1} people, over half of the people in one of the subsets must belong to this party. What happens at the bottom of the recursion? Well, given a set of 2 people, we can ask of one of them if the other is of the same party. So we can envision an algorithm that would return two values, one a yes/no to indicate whether or not over half the people in the subset belong to the same party, and, in case the answer is yes, the other to return one person from that subset who belongs to that party. That algorithm would divide the set of people into two halves, recurse on each half, and check the answers.

   Since we know that over half the people in the overall set belong to the same party, one of the two calls must return yes; if the other returns no, then we can conclude that the person identified in the call that returned yes is the correct answer. The problem arises when both calls answer yes, since in that case the two people identified, call them A and B, could belong to the same party, but could also belong to two different parties, only one of which (obviously) is the correct one. In that case, first test whether A and B are from the same party (one question); if the answer is yes, we are done at this recursion level and can return either one of A or B as the answer—they are both from the majority party. If the answer is no, we now need to verify which of A and B is the correct choice. We do this exhaustively: for each of A and B in turn, we ask, for every person in each subset, whether that person is in the same party as A (resp. B), at a worst-case linear cost in questions. The result is thus a linear-time merging step and the overall running time is the familiar $O(n \log n)$. Note, however, that the algorithm is likely to run considerably faster in practice.

5. What can you do for the previous problem if some fixed (small) percentage of the people are systematic liars, who lie every single time? (Assume that the percentage of liars is reasonably small.) And is that similar to handling some small percentage of errors in the answers, this time distributed evenly among all answers?

   This turns out to be a very interesting and very tricky question, and an opening into a lot of current algorithmic research for various online systems (reputation, auction, etc.). But it is also an old question addressed in the context of communication under noise, trusted communication in the presence of adversaries, etc. Literature on this problem goes back at least to the early 1950s—see the review by A. Pelc in Theor. Comput. Sci. 270 (2002), entitled "Searching games with errors: fifty years of coping with liars"—great title, no?

The short of it is that, if the total number of lies is $O(1)$, then the majority party can still be identified without error in linear time—one simply has to add a few more questions (the number depending on the max number of lies, typically the number grows quadratically with the number of lies) to test correctness. Our D&C algorithm, of course, takes longer, but it, too, can be modified to work with $O(1)$ lies. You can easily devise a strategy for simple cases, even for cases where liars do not always lie, but switch from truthful to deceiving answers at their whim.

The problem gets harder when the number of lies is not explicitly bounded. We then get close to the area called "Byzantine agreement," so named because Byzantine generals, towards the end of the Eastern Roman empire (say, 13-14th century), were notorious for switching factions at the drop of a pin, so that their statements could never be really trusted. In particular, they completely mistrusted each other and a set of such generals would have great difficulty achieving agreement. Byzantine agreement today is a crucial problem in distributed computing and reliable computing. Various solutions have been proposed, none optimal; all rely heavily on randomization to break symmetry. But this would take us much too far for a homework exercise.