**Problem 1 - Quality vs. Price**

This algorithm is actually fairly simple, assuming all numbers are distinct:

Merge sort the list of $(p_k, x_k)$ pairs from minimum price to maximum price.
$qmax = x_1$
$i = 1$
$S = \emptyset$
**while** $i \leq n$ **do**
  **if** $qmax > x_i$ **then**
    $(p_i, x_i) \notin S$
  **else**
    $(p_i, x_i) \in S$
    $qmax = x_i$
  **end if**
  $i = i + 1$
**end while**

Clearly, this runs in $O(n*lgn)$, as the merge sort operation is the bevy of the cost here, taking $O(n*lgn)$. All the other operations are just a single comparison ($O(1)$) for a list of length $n$ - making $O(n)$. $O(n)+O(n*lgn)$ is, of course, $O(n * lgn)$ - I think I'm justified in just stating this claim but if this needs to be proven conclusively note that $\frac{n}{n*lgn}$, or to simplify, $\frac{1}{lgn}$, as $n \to \infty$ is 0. $n + (n * lgn)$ is thus $\leq 2 * n * lgn$, or $O(2 * n * lgn) = O(n * lgn)$.

As for showing that this works - we can see immediately that the first item, $(p_1, x_1)$, has the lowest price due to the merge sort. This means, automatically, that the lowest price hotel cannot be dominated: all other hotels have a greater price and thus cannot have a lower price and a higher quality (which are the two conditions necessary for domination). Thus, the lowest price hotel *must* be a reasonable choice - $(p_1, x_1) \in S$.

What does this mean for the other hotels? All hotels aside from the lowest-cost one have a higher cost - $p_k \geq p_1$ - and thus meet one necessary but not sufficient condition for being dominated by the lowest-cost hotel. To avoid being dominated, all hotels except for the lowest-cost hotel must then have a higher quality than the lowest-cost hotel. If all we were asking was that question, we would be done. But there's a bit of an issue with this, which is that if some $(p_k, x_k)$ exists such that $p_{k+1} > p_k > p_1$ and $x_1 < x_{k+1} < x_k$, then hotel $k$ dominates hotel $k + 1$, but 1 does not dominate $k + 1$. While we move from left to right, price increases monotonically. The "standard" for reasonability for quality must also rise monotonically for no hotels to be dominated, which is why the assignment of new $qmax$ occurs in the algorithm. Let's take a look at that.

For any hotel beyond the first, it is the case that if some previous - and thus lower-price - hotel had a higher quality than the given hotel, it is no longer a reasonable choice, because there is establishedly a hotel with higher quality and lower cost. It thus must be the case that for any hotel to be a reasonable choice, all the hotels of a lower price - all of which came before in the sorted list - must have also had a lower quality. Some of these may have been unreasonable choices - ones that failed the previous hotels = lower quality condition that I just established - but aside from that, the current hotel *must* have a higher quality than the last accepted reasonable hotel - which will have been the one with the highest quality up to that point. This is essentially induction with gaps - the quality increases monotonically with the price for all hotels that are reasonable, and any hotels that would break this monotonicity are not reasonable.

**Problem 2 - Matrix multiplication**

Clearly, if we were to take the naive route here and attempt to just fully multiply the matrix by the vector like a normal person, it'd take at least $O(n^2)$ - disregarding any previous steps calculating the actual factors inside the matrix. Whatever algorithm is created must include the parts of the vector to be multiplied into the matrix at its most basic step - else we end up with an unacceptably-timed algorithm. The recursive structure of the matrix is the key - we can, using our algorithm, reduce the problem first to $n/2$ problems of size 1 - solving half the problem - the a's and b's, and using those, the c's and d's - I'll explain - then returning those answers, moving up a level, and repeating. Here's the general frame of the algorithm.

1. Define a function that splits the input up, taking the vector $x$ as input as well as the number of rows being examined, initially $m$.

a. If the number of rows being examined is 2,

I. Initialize a list of length $n$, with an additional value $k = 1$. Calculate for every other entry in the vector $1 \leq j \leq n$ the following: $(a * x(j), b * x(j+1))$ - and store this in the $k$th position in the list - as well as $(c * x(j), d * x(j+1))$ - and store that in the $\frac{n}{2} + k$th position. Increment $j$ by 2 and $k$ by 1. When this is finished, (i.e. when $\frac{n}{2} + k > n$, or once all indexes have been filled) return the list to the calling function.

b. If the number of rows is greater than 2,

I. Call the function recursively on the number of rows divided by 2. This call shall return a list.

II. With that returned list $\chi$, initialize a new list of length $n$ with index $k = 1$ - the same size - and calculate for every other entry in $\chi$ for $1 \leq j \leq n$: $a * \chi(j) + b * \chi(j+1)$ and store this at the $k$th position in the new list, as well as $c * \chi(j) + d * \chi(j+1)$ and store this at the $\frac{n}{2} + k$th position. As before, increment $j$ by 2 and $k$ by 1 for each iteration of this step.

III. Return this new list to the calling function. If the number of rows being examined is the full $m$ - i.e. the entire matrix is being examined - the returned list with size $n$ is the product of the matrix and the vector.

Okay, let's step back and take a look at that. Essentially, what the algorithm does is to halve the number of rows being looked at, which is $lgn$ 'steps' to go through, and for each individual 'step', starting with the simple products of $a/b/c/d$ and their corresponding vector argument, multiplying results from the previous step together to effectively introduce common factors, and then adding certain results with their neighbors to prepare for the addition of more common factors in the next step. Each 'step' calculates, using values from the previously issued list, $a*$ the leftmost value + $b*$ its right neighbor, as well as $c*$ the leftmost value + $d*$ its right neighbor, so on and so forth $n/2$ times. 4 calculations $n/2$ is $2n$ calculations. $2n$ calculations $*lgn$ steps is in the bounds of $O(nlgn)$.

Why does this work? The algorithm relies on the recursive structure of the matrix - the top left corner of the matrix will always be $a^n$ and multiplied by $x(1)$, its neighbor to the right will always be $a^{(n-1)} * b$ and multiplied by $x(2)$, its neighbor below will be $a^{(n-1)} * c$ multiplied by $x(1)$, so on and so forth. The operative point here is essentially that there is symmetry in the structure - the fact that the matrix is defined

$$\begin{bmatrix} a * H_{k-1} & b * H_{k-1} \\ c * H_{k-1} & d * H_{k-1} \end{bmatrix}$$

essentially guarantees that there is, at all levels of organization, a matrix of the same form to be found. At the very bottom end of this recursive structure, $H_1$ will be patterned throughout, meaning that the top left corner and its three neighbors must have an $a$, a $b$, a $c$ and a $d$ in their factors distributed as indicated above - and so will the cluster of four numbers to the left, and the cluster of four numbers below, so on and so forth. This can be exploited. The matrix being multiplied by the vector in the form $H(i,j)x(j)$ means that all of the matrix's columns, so to speak, share the same index from the vector (e.g. the fourth matrix's column will have each of its terms multiplied by the fourth index from the vector; exactly one of these items in the resultant column will go into each of the sums for the final matrix-vector product). This, combined with the observation above that there are 2-by-2 squares layered end to end all over the complete matrix (as it can be devolved into a series of multiplications ending with $H_1$) indicates that for all pairs of rows, the summation of the first row will begin with some $a * x(1) + b * x(2)$ - each multiplied by a bunch of other stuff in higher level matrices, of course, but these are guaranteed to be factors of the first two terms in such a sum. The second row, comparably, will begin with $c * x(1) + d * x(2)$ - again, each multiplied by other things, but guaranteed each to be factors of the first two terms in the final sum. Why does this matter?

Simply put, the fact that at the bottom level, the factors of $H_1$ are present in a distributed manner means that repeated factors implemented at a particular level can be represented implicitly.

That is to say, the first column of all matrix-vector products has a factor of $a * x(1)$ in its odd-numbered rows, and a factor of $c * x(1)$ in all its odd numbered rows. The same is true for $b$ and $d$ with regard to $x(2)$, in column 2. In column three $a$ and $c$ are multiplied by $x(3)$, so on and so forth. The upshot of this is that if each pair of $a, b$ terms and $c, d$ terms are summed in a linear manner in the first two rows and stored, then you end up with $a * x(j) + b * x(j + 1)$ - which is a factor of the sum of those two partnered columns in an odd-numbered row - which can then be used to represent a single term in the next level up. This last fact is due to the fact that any 2 by 2 'bottom level' matrix can have its $a$ and $b$ terms summed and *consistently* say that they'll be in the same quadrant of the matrix of the next level up - for example, once in the first row $a * x(1)$ and $b * x(2)$ are summed, when trying to analyze the 4 by 4 matrix that contains the 2 by 2, these terms will then have a common factor - in this case, $a$ (as 1 and 2 are both $\leq 4/2$). They can then be represented as one term, multiplied by $a$, and summed with the $b$ result from the neighboring 2 by 2 quadrant, so on and so forth.

The recursive structure of the matrix thus justifies the calculation of halves - the first 2 rows, the first 4, the first 8, so on - to calculate the values of the other halves. Each of these steps can be performed in $O(n)$, and naturally, there are $lg(n)$ of them.

**Problem 3 - Finding the Bottom**

**a.** The algorithm here is fairly simple.

Input: $x$, the set of lines
$maxval = -\infty$, $maxes = \emptyset$
**for** each line $i$ in the set of lines **do**
  $y = a_i * x + b_i$
  **if** $y > maxval$ **then**
    $maxes = i, maxval = y$
  **else**
    **if** $y = maxval$ **then**
      $maxes = maxes \cup \{i\}$
    **end if**
  **end if**
**end for**
**if** $|maxes| = 1$ **then**
  **if** $a_i > 0$, that is, the slope of the maximum-y line is positive **then**
    $x^* < x$. The solution is to the left of x.
  **else**
    $x^* > x$. The solution is to the right of x.
  **end if**
**else**
  **if** for all $i \in maxes, a_i > 0$ **then**
    $x^* < x$. The solution is to the left of x.
  **else**
    **if** for all $i \in maxes, a_i < 0$ **then**
      $x^* > x$. The solution is to the right of x.
    **else**
      $x* = x$. The solution is at $x$.
    **end if**
  **end if**
**end if**

I probably shouldn't have written this all out in algorithmic form because now it looks really confusing, but essentially the structure of the algorithm is that you step in a linear manner through the collation of lines, seeing if a particular line has the highest $y$ value for the input $x$. If there's only a single line at the highest $y$, the solution is either to the right, if the slope of the line is negative (that is to say, dropping a ball at $(x, \infty)$ would have it roll to the right), or to the left, if the slope is positive (dropping the ball would make it roll to the left).

On the other hand, if there's more than one line at that point - which is to say, $(x, y)$ is an intersection of multiple lines - the algorithm picks up and adds to a set what's up to that point the highest $y$ - then we're faced with a different question. If all the lines intersecting at this point have positive slopes, then the solution is to the left. Dropping a ball here would roll it, however gradually, to the left, as the highest slope a positively-sloped line can have is anything short of vertical. Similar to the depiction in Figure 2 of the problem, one of the lines can overpower the others if all of their slopes face the same direction. Symmetrically, if all the lines intersecting at this point have negative slopes, the solution is to the right, for the same reason. The real catch here is if there's any lines with differently signed slopes - at least one positive and at least one negative - in which case, this is the solution. The intersection would be a point at which a ball dropped from infinity would rest - as it can't 'roll' left or right - and it is also the highest such intersection of points in the collection of lines, as established above.

This clearly runs in $O(n)$ - it merely runs through each line once, and in the worst case scenario, all the lines intersect at one point, and need to be added to the maxes set - in which case it's another simple $n$-element pass to check if there are any signs differing between the maxes lines. It also fairly simply can be proven to work - it can be taken for granted that if there's a solution then it *must* be at an intersection of at least one positively sloped and at least one negatively sloped line, because if we were to in contradiction posit a solution that was 'above all lines' just by sitting on/next to a lone line, we could show that moving it down along the line's slope (right for negative, left for positive) we would get a better solution. The only way to stop here is at an intersection of lines. Anyway, that small excerpt basically nixes the lone $y$ case - the analogy of the ability of the ball to continue rolling is helpful here - and more or less takes care of the case in which all maximum lines have the same

sign. This, too, is because if they all have the same sign, the point can just be moved down along their (successive) slopes to get a better solution. The only trouble about showing correctness comes along when a group of maximum lines have differently signed slopes. The fact that they're even in consideration means that the y-value for all other lines for the given x was smaller - which wouldn't immediately rule anything out, but assuming we put the point at the maximum y value, there's no way for it to move right or left. It is, so to speak, stuck by the intersecting differently-sloped lines.

**b.** As for the actual full algorithm, solving the problem should essentially look similar to the quick-select algorithm we discussed in class:

1. Randomly or arbitrarily pair up all the lines in the collection of lines. (They're presented in no particular order, so it might suffice to just put each one with its neighbor. Not fully sure, though). If there's an odd number of lines it should be fine to add an arbitrary horizontal line at $x = -\infty$ or $x = \infty$ and intersect the odd one out with this line - the other lines are otherwise guaranteed to form intersections at more central $x$ values. The pseudo-line here will never have the median intersection point $x$ value (as it'll be at infinity in one of two directions), and thus the pseudo-line will be thrown out by step 3 (assuming we place it at positive infinity if we find that the solution is left of the median and negative infinity if the solution is right of the median).

2. Calculate the intersection for each pair using the equation $x = \frac{b_j - b_i}{a_i - a_j}$, and then plugging that $x$ value into either of the lines' equations to get a $y$.

3. Get the median $x$ value and call the procedure described above in 3a on that value. Depending on the result...

a. If $x^* > x$, the solution was to the right of the median. Take all the intersection points to the left of and including $x$, and for each pair, discard the line with the lower (i.e. more negative) slope.

b. If $x^* < x$, the solution was to the left of the median. Take all the intersection points to the right of and including $x$, and for each pair, discard the line with the higher (i.e. more positive) slope.

c. If $x^* = x$, the solution is at $(x, y)$.

4. If the solution hasn't been found, repeat sans the discarded lines.

Interestingly, this actually does run in $O(n)$. Making $n/2$ random pairs and doing a constant time calculation for each of them is in the $O(n)$ bound, the call for a median is in the $O(n)$ as well, as is the call for 3a's left/right procedure - for a total of 3 n-time steps. The catch is that there's a recurrence. However, a median $x$ call on $3a$ will either find the solution, nix half of the lines whose intersections are to the right, or nix half of the lines whose intersections are to the left. As this is the median, half-ish of the intersections of random pairs should be to its left and half-ish should be to the right. To discard one of each of these - a quarter of the full number of lines - is to have the recurrence working with $3/4 * n$ where $n$ is the current number of lines. $T(n) = T(\frac{3}{4}n) + 3n$ is the recurrence we're looking for. For the first call, then, $T(n) = T(\frac{3}{4}n) + 3n$, then one level down $T(n) = 3n + \frac{9}{4}n + T(\frac{3}{4} * \frac{3}{4}n)$, so on and so forth. This results in $O(n)$ complexity - the $T(3/4 * n)$ term is a convergent series. Specifically, the sum of $3 * (.75)^x$ from $x = 0$ to $\infty$ is 12 (thank you, WolframAlpha). $T(n) = 12n$ or something like it, in that case, and thus the algorithm is in $O(n)$.

The reason that this works is actually fairly simple and again, looks a lot like an algorithm like quick-select or quicksort. We know that we can calculate easily the pairs of intersections, and that we can pick one of them out as the median and decide whether or not the solution is to its left or right; this much we confirmed in $3a$. The reason the discarding of half of the lines on one side or the other works is essentially because of the condition that the solution must be *above* all lines: thus, if the solution is to the right of a given intersection, whichever line's slope has a lower value will be 'below' the higher-sloped line at all points to the right - and thus won't matter when it comes to playing a role in the solution itself. The same applies for the case where the solution is to the left of the given intersection, and of course, when the solution is *at* the given intersection the algorithm can just stop there. Lastly, we do know that the algorithm will eventually find a solution - I showed above that if we consider that 3 full passes over $n$ are necessary for the first recurrence (which seems a good estimate at very least), the fact that 1/4 of the lines remaining are removed each time the procedure recurs means that we end up with a linear-time algorithm - as it continues to economize on the number of elements it views each recurrence by keeping about 3/4 of them, which must eventually reduce to at least 2 lines. It's entirely possible to accidentally find the solution before reducing all the way, of course, but in the worst case, only one intersection will remain after the final recurrence call - and this must thus be the solution.