Problem 1. (a) We have four coordinates x, x', y', y' for the query rectangle $[x : x'] \times [y : y']$ representing our search space, and we also have four coordinates a, a', b, b' for every rectangle with dimension $[a : a'] \times [b : b']$. We know that in order for a rectangle $[a : a'] \times [b : b']$ to be contained in a query space $[x : x'] \times [y : y']$, we must have $x \leq a < a' \leq x'$ and $y \leq b < b' \leq y'$. Since $a < a'$ and $b < b'$ by default (a rectangle must have non-zero volume), this is equivalent to checking $(a, a', b, b') \in [x : x'] \times [x : x'] \times [y : y'] \times [y : y']$. In other words, we transform the problem into a range search problem in $\mathbb{R}^4$.

Next, in terms of data structures, we can use the approach from class to construct a 4-dimensional range tree by showing that the $O(n \log^{d-1} n)$ space and $O(\log^d n + k)$ query time complexities hold for any d via induction. For the 1-dimensional base case, a binary search tree naturally uses $O(n)$ space for n points and $O(\log n + k)$ time to find the left range bound and then walk to the right range bound. Then, given a d-dimensional range tree data structure with $O(\log^d n + k)$ query time and $O(n \log^{d-1} n)$ space, we want to show that we can construct a $(d+1)$-dimensional range tree with query time $O(\log^{d+1} n + k)$ and space complexity $O(n \log^d n)$.

- Time: suppose we are constructing a BST on the first coordinate, i.e. x. We can split x's range into $O(\log n)$ range subintervals, such that each internal node corresponds to a d-dimensional range tree. Then, it takes $O(\log n)$ time to find the subtrees within the search interval, and then $O(\log^d n)$ time to recursively range search each of the d-dimensional subtrees, for a total time of $O(\log^{d+1} n)$. Lastly, once the left-most point (i.e. rectangle) has been found, it takes an additional k steps to walk to the right-most rectangle within the bound, so the overall runtime is $O(\log^{d+1} n + k)$.
- Space: similarly, we construct a BST on the first coordinate which takes $O(n)$ space for n internal nodes as well as additional space for each of the d-dimensional subtrees. Note that for each $i < \log n$ (i.e. level of the outer tree), there are 2^i subintervals of length $n/2^i$ (and therefore size $O(n/2^i \log^{d-1} n/2^i)$), so we have

$$\sum_{i=1}^{\log n} 2^i \, O\left(\frac{n}{2^i} \log^{d-1} \frac{n}{2^i}\right) \leq \sum_{i=1}^{\log n} O(n \log^{d-1} n) \leq O(n \log^d n)$$

  Thus, the overall space complexity of the $d+1$ dimension tree is $O(n + n \log^d n) = O(n \log^d n)$.

Thus, we have constructed a $(d+1)$-dimensional range tree with query time $O(\log^{d+1} n + k)$ and space complexity $O(n \log^d n)$ from a d-dimensional range tree data structure, closing the induction. Using this result, we know that a 4-dimensional range tree can solve the rectangle problem using $O(n \log^3 n)$ space and $O(\log^4 n + k)$ time for queries.

(b) Note that if a polygon fits in a query rectangle, then its smallest containing rectangle also fits in the query rectangle; this is because the smallest containing rectangle corresponds 1-to-1 with the four most relevant dimensions of the polygon. We can convert each polygon into its smallest containing rectangle by taking a list of points $\{(x_1, y_1), (x_2, y_2), \ldots\}$ and extracting the rectangle formed by $(x_{min}, x_{max}, y_{min}, y_{max})$ in linear time. Afterwards, we can use the algorithm from part (a) on the smallest containing rectangles to determine the solution. Thus, the polygon problem can be solved using $O(n \log^3 n)$ and $O(\log^4 n + k)$ query time with a one-time pre-processing cost of $O(n)$.

Problem 2. Our rotation-sweep algorithm is as follows:

1. Sort the wall endpoints in order of the angle they form with the player's horizontal (i.e. if the player is facing east, then a point straight ahead of him is at 0°, and a point directly to the left of where he is standing is at 90°). If two wall endpoints have the same angle, order them by shortest distance to the player, which is well-defined since no wall segments cross. This will guarantee that we process wall segments that are closest to the player first.

2. Initialize a binary search tree, which will store the wall segments that are currently being processed by the sweep radius by their distance from the player. Note that because we have non-crossing wall segments, although the distance from the wall to the player may change depending on which endpoint is being used to calculate the distance, the overall ordering of wall segments within the BST is still invariant (two walls along the same arc will never intersect). Thus, we can simply use the distance from the starting endpoint of the wall segment to the player.

3. Iterate through the sorted list of wall endpoints, performing a counter-clockwise "sweep" around the player. For each endpoint:

   - If this endpoint is the start of a wall segment (i.e. the first encounter when sweeping), insert the wall segment into the BST with its current distance from the player, calculated from the start endpoint. If this insertion distance is the minimum distance within the tree, mark the wall segment as visible.
   - If this endpoint is the end of a wall segment (i.e. the second encounter of a wall segment), find and remove the segment from the BST. If this segment's distance was the minimum distance in the tree, mark it as visible. Also, mark the new minimum as visible.

   As stated previously, since we have non-crossing wall segments, at any time the minimum distance within the BST corresponds to a visible wall segment. Thus, whenever we insert or remove a segment from the BST, we want to update the current (and newly) visible segments.

   Thus, the algorithm will perform a 360° sweep around the player and determine which walls are visible to the player. Step 1 takes $O(n)$ time to pre-process each wall segment and calculate the endpoint angles and distances, and $O(n \log n)$ time to sort the endpoints in order. Step 2 takes $O(1)$ time. Step 3 takes $O(\log n)$ time for each insertion/deletion of an endpoint in the BST, and $O(1)$ time (by keeping a pointer) to find the minimum and mark it as visible, so the total is $O(n \log n)$ since there are $O(n)$ endpoints. Thus, the overall runtime of the algorithm is $O(n \log n)$.

Problem 3. (a) If p is a member of the closest pair behind the sweep line, then this pair must be a distinct pair with distance d' that is less than the previous closest pair, i.e. $d' < d$. This implies that the other point must lie somewhere within the semicircle with radius d centered around p, which is a subset of the strip region that, by definition, is quite close to p.

(b) We know that by definition, the previous closest pair had distance d. Thus, if the new point p is part of the new closest pair, then the other point must lie within the portion of the strip that is a semicircle centered around p with radius d. Moreover, we know that any points within this portion (not including p) must be at least d distance apart, otherwise they would be the previous closest pair. We can see that geometrically, it is possible to fit at most 3 points into this semicircle that are each at least d distance apart from all the other points: one such configuration would be placing one point at the top of the semicircle, one at the bottom, and one somewhere at the very left. If there were any more than these 3 points in the semicircle, then at least two points would be within d distance of each other. Thus, the semicircle portion of the strip can only contain a constant number of non-new points.

(c) *Note: to make things slightly easier, instead of considering the semicircle portion of the strip from part (b), we will use the rectangle that circumscribes the semicircle with dimensions* $2d \times d$. *Although there is some redundant area within this portion –* $d^2(2 - \pi/2)$*, to be exact – where points that are not candidates for the new closest pair may be, it will allow us to more efficiently utilize the data structure below. There can be at most* 6 *non-p points in this rectangle portion that are d distance apart from each other: for instance, one at each of the corners, and two at the center-left and center-right midpoints.*

We can use a balanced binary search tree to store the set of points that are presently contained within the strip, indexed by their y-coordinate (but each node still contains the x-coordinate information). While performing the sweep, whenever the sweep line encounters a new point, it inserts it into the BST and performs the following operations:

- Find closest pair: first, insert the new point p with y-coordinate y_p into the BST, which takes $O(\log n)$ time since there can be at most n points within the BST at any given moment. Then, walk the tree to the left and right until $y_p - d$ and $y_p + d$ to gather the list of points in the rectangle portion of the strip, which will take $O(1)$ time since there are a constant number of points in the portion. Lastly, calculate the Euclidean distance between these candidate points and the new point, and update the closest pair if a new minimum is found.
- Delete points outside the strip: first, pre-process all the points to get a list of points sorted by x-coordinates. This only needs to be done once, and costs $O(n \log n)$. Next, we need to delete any points with x-coordinate less than $s - d$, where s is the current location of the strip line. To do this, we can binary search the list to get the first point with $x < s - d$, then continuously delete points to the left until we reach a point that has already been deleted previously (since the points are stored in order, it means that there are no other points that need to be deleted). The total cost of deletion is amortized $O(n \log n)$ for the entire algorithm, because each point only gets deleted once. Thus, the only other cost is the initial binary search, which takes $O(\log n)$ time per iteration, or $O(n \log n)$ in total for n points added to the BST.

Thus, since insertion and find closest pair takes $O(\log n)$ time for n items, and deletion takes amortized $O(n \log n)$ time over the course of the algorithm, this data structure determines the closest pair in $O(n \log n)$.

(d) We can generalize the algorithm to any higher dimension k by sweeping over the last dimension, i.e. hyperplane-sweep. As before, we know that behind the sweep hyperplane, there is a strip of width d containing the next closest pair. Any time we encounter a new point while sweeping, we must perform the same find-closest-pair step and deletion of points outside the strip as in part (c). Using a $k-1$ dimension range tree, we can perform range queries within the strip in $O(\log^{k-1} n + j)$ time, where j is the maximum number of points within the k-dimensional half-cube that constitutes the "candidate" portion of the strip. Similarly, the cost of deletion from the range tree is amortized to $O(\log^{k-1} n)$, and the range tree itself uses $O(n \log^{k-2} n)$ space. Thus, since the hyperplane-sweep encounters n points, the overall time bound is $O(n \log^{k-1} n)$.

(e) The Voronoi diagram partitions the 2-dimensional space into cells, and it follows that the closest pair of points corresponds to two adjacent cells. Since each pair of adjacent cells corresponds to a single edge, it suffices to find the edge that is closest to both of its neighboring cell vertices. We showed in class that the number of edges in the Voronoi diagram is bounded by $O(n)$, so determining the closest pair takes linear time, since we just need to iterate over the list of edges. Finally, pre-processing the points and constructing the Voronoi diagram itself takes $O(n \log n)$ time, so this is a valid alternative algorithm in 2 dimensions.

Problem 4. (a) Our algorithm is as follows:

Visit the following points in order: $1, -2, 4, -8, 16, -32, \ldots$

In other words, we double the distance traversed in a single direction each time before reversing, such that the i-th turning point is $(-2)^{i-1}$. Note: we can also start from -1 and proceed to $2, -4, 8, \ldots$ without loss of generality by flipping signs, as the analysis that follows only depends on the "worst case" scenario in which the destination lies just beyond a turning point.

The following table lists the first few turning points, distance travelled before turning, and total distance:

Turning Point	Steps Before Turn	Total Steps
1	1	1
-2	3	4
4	6	10
-8	12	22
16	24	46
-32	48	94
64	96	190

Next, we list the total distance travelled to find the "worst case" bridges, which are located one step beyond each turning point, for the first few turning points:

Bridge Location	Total Steps
2	8
-3	17
5	35
-9	71
17	143

As the table shows, in general, for the i-th turning point, it takes $9 \cdot 2^{i-1} - 1$ to reach the "worst case" bridge, which is located one step beyond the turning point at $|2^{i-1}| + 1$ distance away from the origin in magnitude (corresponding to the OPT distance travelled). Thus, we have

$$\frac{9 \cdot 2^{i-1} - 1}{|2^{i-1}| + 1} \leq 9$$

so this algorithm is 9-competitive.

(b) Suppose that instead of changing direction each time we reach a turning point, we instead flip a fair coin and continue in our original direction if it lands on heads, and turn around if it lands on tails. For instance, a possible sequence of visits could be: $1, 2, 4, 8, -16, -32, 64, 128, \ldots$ and so on.

Consider again the "worst case" scenario, in which the bridge is located one step beyond a turning a point. Now, with the randomized coin toss, we have a $1/2$ chance of repeating the results of the deterministic algorithm, which has a maximum distance of 9-times OPT. On the other hand, we have a $1/2$ chance of continuing in the same direction, which would allow us to find the bridge with just one additional step as shown in the following table:

Turning Point	Bridge Location	Total Steps to Bridge
1	2	2
-2	-3	5
4	5	11
-8	-9	23
16	17	47
-32	-33	95
64	65	191

In other words, for the i-th turning point, it would take $3 \cdot 2^{i-1} - 1$ steps to reach the corresponding bridge (assuming the coin toss lands on heads). Since $\frac{3 \cdot 2^{i-1}-1}{|2^{i-1}|+1} \leq 3$, we have that the expected distance of the randomized algorithm is

$$\frac{1}{2} \cdot 9 + \frac{1}{2} \cdot 3 = 6$$

so this randomized algorithm is 6-competitive.

Problem 5. (a) Regardless of the weight assignments, in order for the online algorithm to be wrong, there must be cumulatively at least 1/2 weight voting for the wrong answer. After realizing that the answer is wrong, we will halve their weights, so the overall total weight is decreased at least 1/4. This leaves the faculty with at most 3/4 remaining weight, hence the decrease by a factor of 4/3. Supposing that there are k mistakes, there will be $n * (3/4)^k$ remaining weight spread across the faculty in the end.

(b) The wisest faculty member starts with weight 1 and makes m mistakes, each of which halves their remaining weight. Thus, the wisest faculty member is left with $1/2^m$ weight in the end.

(c) Combining parts (a) and (b), we see that the remaining weight at the end is lower bounded by $1/2^m$ in the best case and $n * (3/4)^k$ in the worst case, so we have:

$$\begin{aligned}
\frac{1}{2^m} &\leq n\left(\frac{3}{4}\right)^k \\
m \log \frac{1}{2} &\leq k \log \frac{3}{4} + \log n \\
-m - \log n &\leq k \log \frac{3}{4} \\
\frac{m + \log n}{\log \frac{3}{4}} &\geq k \\
2.41(m + \log n) &\geq k
\end{aligned}$$

Thus, we have $k \leq 2.41(m + \log n)$ is the upper bound of the number of mistakes we make, and therefore we are 2.41-competitive against the wisest faculty member, proving the claim.

(d) Regardless of the weight before the question, the weight portion of faculty with the wrong answer is F, so the total weight decreases to βF. Similarly, the weight portion of faculty with the right answer is $1 - F$, and this quantity does not change (since they were correct). Thus, if W is the total weight before, then we have

$$W' = (\beta F + (1 - F))W = (1 - (1 - \beta)F)W$$

Thus, the old total weight is multiplied by a factor of $(1 - (1 - \beta)F)$ to obtain the new total weight.

(e) The expected number of wrong answers is given by $\sum_i F_i$ where F_i is the fraction of weight of faculty with the wrong answer on question i (since $E[\text{wrong}] = F_i$). Similarly to parts (a) and (b), we have that the total weight in the end is upper bounded by $n \prod_i 1 - (1 - \beta)F$, and lower bounded by the wisest faculty member, who is left with β^m weight in the end. Thus, we have

$$\begin{aligned}
\beta^m &\leq n \prod_i 1 - (1 - \beta)F \\
m \log \beta &\leq \log n + \sum_i \log(1 - (1 - \beta)F) \\
m \log \beta &\leq \log n - \sum_i (1 - \beta)F \qquad \text{since} \log(1 - z) = -z \\
\sum_i F &\leq \frac{\log n - m \log \beta}{1 - \beta}
\end{aligned}$$

So the expected number of wrong answers is at most

$$\frac{\log n - m \log \beta}{1 - \beta} = \frac{m \log(1/\beta) + \log n}{1 - \beta}$$

(f)