1. (a) Select the points \((\frac{i}{2n}, 0)\) for \(0 \leq i < n\), in that order. I claim that coloring the \(i\)th point colors at least \(\frac{1}{2}\) of the square. Indeed, consider the rectangular region \(R\) bounded by \((\frac{1}{2}, 0)\) and \((1, 1)\). Since all points are along the \(x\) axis, by the triangle inequality selecting \((\frac{i}{2n}, 0)\) will color all points with \(x > \frac{i}{2n}\). In particular, this will color \(R\). So, we use at least \(\frac{n}{2} = \Omega(n)\) paint.

(b) We use backward analysis. Suppose there are currently \(m\) points left. Let \(S\) be the expected size of the region repainted when the last point is removed. Note that \(S\) is also the expected size of the region corresponding to the point removed, where the expectation is over which point is removed.

Let the events of removing point 1, point 2, etc. be \(X_i\). Then notice that

\[
E[S] = \sum_{i=1}^{m} E[S | X_i] P\{X_i\} = \sum_{i=1}^{m} \frac{1}{m} E[S | X_i]
\]

Notice that \(P\{X_i\} = \frac{1}{m}\) by a symmetry argument: this is a random permutation.

Now let \(E[S | X_i] = x_i\). Notice that \(\sum_{i=1}^{m} x_i = 1\) because the Voronoi coloring covers the whole square. So \(E[S] = \frac{1}{m}\).

Hence by Linearity of Expectation and backward analysis, the total expected amount of paint is

\[
E[S_1] + E[S_2] + \ldots + E[S_n] = \sum_{i=1}^{n} \frac{1}{n} = H_n \leq \log n + \gamma \in O(\log n)
\]
2. The key idea is to use a sweepline. Consider for each rectangle the left edge and right edge. We sweep from left to right: when we encounter a left or right edge we continue the edge until we either hit the boundary of \( S \) or another rectangle in \( R \)'s. This splits up \( S \) into rectangular regions.

Now let’s prove the desired bounds. First note that extending a left edge can only create one new region, and extending a right edge can create at most two new regions (below and above the rectangle). So, there are at most \( 3m \) regions created. We get an extra 1 because there may be another region to the right of the last right edge.

Now let’s justify the runtime. To do the sweepline, we sort the left and right edges by \( x \) coordinate, which is \( O(m \log m) \). We will also need to be able to lookup the closest bottom or top edge at an \( x \) coordinate. We do this in a preprocessing pass as follows. While sweeping from left to right, we keep track of the “open” left edges (i.e. rectangles that haven’t hit the right edge), and two \( y \) coordinates: Whenever we hit a new left edge we add it to the open set and update the \( y \) coordinates. These store the lowest open bottom edge above the rectangle and the highest open top edge below the rectangle, respectively. When we hit a right edge, we similarly set set the \( y \) coordinates, and remove the “open” left edge. These coordinates all start out initialized to \((n, 0)\) (top and bottom, respectively).

Now, lookup at a left or right edge for the top and bottom is \( O(\log m) \) (just store them in an ordered set). So, we can tile \( S \) in time \( O(m \log m) \) after the sorting (preprocessing is also \( O(m \log m) \)). Finally, we need to be able to get \( R' \). Note that we can store (even more) information: when extending an edge \( e \), suppose we hit some edge \( f \). Then we can set \( f \) to store the rightmost line which hits it.

So, when we extend a right edge to “close” a region, we can in \( O(\log m) \) compute the entire boundary of the new rectangle \( r \). It will be the edge extension, the rightmost line along the (bottom or top) of the rectangle we extended from, the edge we hit when extending, and the (bottom or top) edge itself.

When we extend a left edge, the argument is much simpler: we just need the whole edge extension, the two edges it hits from top and bottom, and the previous line we encountered during sweepline.

So, computing \( R' \) only takes \( O(1) \) additional work while sweeping, so the overall sweep time is \( O(m \log m) \) and we are done as we get \( O(m \log m) + O(m \log m) = O(m \log m) \).
3. (a) We construct two hash tables, one for red and one for green points. For a point \( x \), we hash it into the bucket corresponding to \( \lfloor x/\alpha \rfloor \). In particular, we store the minimal and maximal point in the bucket. Notice that there cannot be a red \( x \) in bucket \( z \) and a green \( y \) also in bucket \( z \), as this would imply that

\[
\lfloor x/\alpha \rfloor = \lfloor y/\alpha \rfloor \implies |x/\alpha - y/\alpha| < 1 \implies |x - y| < \alpha
\]

which is contrary to the statement.

So, iterate over the green buckets. For a bucket \( z \), consider the minimal green point in \( z \) and the maximal red point in \( z - 1 \). If this distance is \( \alpha \), we are done. Else, continue iterating.

Do the same for the red buckets. Note that in the first, \( \mathcal{O}(m) \) iteration we are looking for pairs \((r, g)\) with \( r < g \). In the next iteration we are looking for \((g, r)\) with \( g < r \). This must cover all cases, so this is expected \( \mathcal{O}(m + n) \) because that is also the construction time of the hashtables.

(b) We give an \( \mathcal{O}(i) \) algorithm, where \( i \) is the total number of points in the hashtable. This will be sufficient for the next part, as we will see shortly. Suppose we are looking up a green point \( p \). Then compute \( z = \lfloor p/\alpha \rfloor \). There are three things we look at amongst the red buckets: the max in \( z - 1 \), the min in \( z + 1 \), and the whole bucket \( z \). If there are no elements in \( z \) and the ones in \( z - 1 \) and \( z + 1 \) are greater than \( \alpha \) away, return “NOT CLOSEST”. Doing this costs \( \mathcal{O}(1) \). Else, compute the min over all these elements, of which there are at most \( i \) so the runtime is \( \mathcal{O}(i) \).

(c) We randomly permute the given \( n + m \) points, and insert into the defined grid. To do this, consider a \texttt{Lookup(Grid, p)} call. This costs \( \mathcal{O}(1) \) if \( p \) is not closer than the current min distance, and \( \mathcal{O}(i) \) if it is. If \( p \) is closest, we let \( r' \) be the new closest distance and rehash all the points according to \( \lfloor x/r' \rfloor \). This is also \( \mathcal{O}(i) \). Hence, insertion costs \( \mathcal{O}(1) \) if \( p \) is not the new closest and \( \mathcal{O}(i) \) otherwise.

Now we show that this is expected linear via backward analysis. Suppose we have already inserted all the points. Similarly to inserting, we let the deletion cost \( \mathcal{O}(1) \) if the min distance does not change, and \( \mathcal{O}(i) \) otherwise.

Note that the probability that the min distance does change is at most \( 2/i \). This is because if there are \( g \) green points and \( r \) red points, the probability is at most \( \frac{2}{i} \cdot \frac{1}{g} + \frac{r}{i} \cdot \frac{1}{r} = \frac{2}{i} \) that we will change the min distance. If there is more than one such pair, the probability is lower (similar to Sariel Har-Peled). So, the expected time for one operation is \( \mathbb{E}[T_i] \leq \frac{1}{i} \cdot \mathcal{O}(1) + \frac{2}{i} \cdot \mathcal{O}(i) \in \mathcal{O}(1) \), as desired. Summing over all \( i \) gives the desired \( \mathcal{O}(m + n) \) result.