This is an oral presentation assignment. Again, there are three regular problems (#1-#3) and one programming problem (#4). You should work in groups of three. The sign-up sheet will be online soon (details on Piazza) and your group should sign up for a 1-hour slot. Each person in the group must be able to present every regular problem. The TA/Professor will select who presents which regular problem. You are not required to hand anything in at your presentation, but you may if you choose.

The programming problem will be submitted to autolab, due May 4th, 11:59pm. You will not have to present anything orally for this. You can discuss the problem with your group-mates, but must write the program by yourself. Please do not copy.

(25 pts) 1. **Counting Chord Crossings.** You’re given a unit circle $C$, and $n$ chords on the circle. A chord is a line segment $(a, b)$, where $a \neq b$ and both $a$ and $b$ are on the circle. Give an $O(n \log n)$ algorithm to count the number of pairs of chords that intersect. For simplicity you can assume that no two chords have any endpoints in common.

**Solution:** For each endpoint on the circle, we can calculate its angle which is a number in $[0, 2\pi)$. Sort these angles and re-number those endpoints from 1 to $2n$ in increasing order with respect to the angles. Then, sort the chords according to their smaller endpoint. Now, each chord can be viewed as a pair of integers $[a_i, b_i]$ with $a_i < b_i$. Consider two such chords $[a_1, b_1]$ and $[a_2, b_2]$, where $a_1 < a_2$. They intersect if and only if $b_1 < b_2$. So we can use a sweep-line algorithm, keeping track of the number of “active” chords at the current time. When we encounter the first point $a$ of some chord $[a, b]$, the number of intersections with those chords prior to it is simply the number of currently-active chords with larger endpoints is smaller than $b$.

To compute this number dynamically, we use a SegTree on the $2n$ endpoints. The index $i$ contains 1 if $i$ is the right endpoint of some chord whose left endpoint we’ve seen already. At first, all entries are zero. When we reach point $a$ corresponding to the interval $[a, b]$, we do a range search query on the interval $[a+1, b-1]$, and add the result to our return value. After that, we set the counter for $b$ to 1. When we complete our sweep-line algorithm, we return the accumulated value.

Complexity: the sorting on endpoints and chords takes $O(n \log n)$. In the sweepline algorithm, we sweep on $n$ chords, each chord doing a range query and a single point update, both are $O(\log n)$ according to properties of segment tree. Hence the total time complexity is $O(n \log n)$.

(25 pts) 2. **Algorithms from another Planet.** Your spaceship has landed on LineWorld, which as the name suggests, looks like the real line $\mathbb{R}$. You start at the origin $x_0 = 0$. At each day $t \geq 1$, the adversary puts a linear force field which is centered at some location $c_t \in \mathbb{R}$ and has slope $s_t \geq 0$. This force field means that if you are standing at location $x \in \mathbb{R}$, you spend $f_t(x) := s_t \cdot |x - c_t|$ of fuel for this day. Moreover, you spend 1 unit
of fuel for each unit of distance you move along the line. So on day $t$, you observe the force field for this timestep (i.e., you get to know $s_t, c_t$). Then you may move from the previous location $x_{t-1}$ to some location $x_t$ (it takes you zero time to travel), and then spend day $t$ at this location $x_t$, hence burning $|x_t - x_{t-1}| + f_t(x_t)$ amount of fuel on this day. The total fuel usage until time $T$ would hence be

$$\sum_{t=1}^{T} (|x_t - x_{t-1}| + f_t(x_t)).$$

You cannot see the future, so you make decisions on each day $t$ without knowing the force fields on days $t+1$ or later. You want to give an online algorithm that is 4-competitive. I.e., for each $T$, the fuel usage of the algorithm until day $T$ is at most 4 times the fuel usage of any other algorithm $B$ until that same day.

Here’s an algorithm: on day $t$, move from $x_{t-1}$ towards the center $c_t$, until either (a) you reach $c_t$, or (b) the distance you have traveled (i.e., $|x_{t-1} - x|$) equals $f_t(x)$ at the point $x$ you reach. This is your new point $x_t$. Show this algorithm is 4-competitive.

**Hint:** What would be a good potential to use?

**Solution:** Let $b_t$ be the position of $B$’s server at time $t$. Let $\Phi_t = 2|x_t - b_t|$, and hence the initial potential is zero. We want to show that for each day

$$f_t(x_t) + |x_t - x_{t-1}| + (\Phi_t - \Phi_{t-1}) \leq 4f_t(b_t) + 2|b_t - b_{t-1}|. \quad (1)$$

We first let $B$ change its location from $b_{t-1}$ to $b_t$. This can increase the potential by at most $2|b_t - b_{t-1}|$ which is balanced by the increase in the right hand side by the same quantity. Let us denote the potential at this time by $\Phi_{t-1/2}$.

Then let the algorithm move its location from $x_{t-1}$ to $x_t$. For brevity, define $d_t := |x_t - x_{t-1}|$. In both cases we know that

$$d_t \leq f_t(x_t).$$

Consider the following cases.

- Suppose $f_t(x_t) \leq f_t(b_t)$. The increase in the potential $\Phi_t - \Phi_{t-1/2} \leq 2d_t$. Hence, the LHS of (1) is at most $f_t(x_t) + d_t + 2d_t \leq 4f(x_t) \leq 4f(b_t)$.

- Else $f_t(x_t) > f_t(b_t)$. This means that we moved from $x_{t-1}$ to $x_t$, but did not reach $c_t$ (else we would have stopped there, and would get $0 = f_t(x_t) \leq f_t(b_t)$). In this case we know that $d_t = f(x_t)$. Moreover, we know that we moved closer to $b_t$ (since all points with lower force field are points we have not reached yet), and hence $\Phi_t - \Phi_{t-1/2} = -2d_t$. Hence the LHS of (1) is exactly zero, but the RHS is non-negative, which gives us the claim.

These cases complete the proof.
(25 pts) 3. **Points, Lines, and Triangles.** You’re given a set $S$ of $n$ points in the plane with integer coordinates.

(a) Give an $O(n^2)$ algorithm that finds the largest subset of $S$ all of whose points are colinear.

(b) Give an $O(n^2)$ algorithm to find the triangle among the points of $S$ with the largest area.

**Solution:**

(a) For each point $p$, we find the maximal number of points that are colinear with it in $O(n)$ time. Then we simply take the maximum of all these results, to get the $O(n^2)$ bound.

Indeed, for a fixed point $p$, consider any other point $q$. Since all coordinates are integers, the slope of $q-p$ can be uniquely represented by a pair $(x, y)$ of relatively prime integers with $y \geq 0$. We can insert this pair into a dictionary (keeping track of frequencies). Now the pair $(x, y)$ with the highest frequency gives us the line passing through $p$ with the highest number of points on it. Since every insertion or query on a dictionary entry takes $O(1)$ time (using, say, a hash table), the process for point $p$ takes $O(n)$ time.

(b) First, the three vertices of the result triangle must be on the convex hull formed by the $n$ points. So we first compute their convex hull of $S$, which consists of at most $n$ points. Again, we claim that we can find the max-area triangle for any point $p$ on the convex hull in linear time. Iterating over all points gives us the claimed $O(n^2)$ runtime.

Fix a vertex $A$ on the convex hull, and number all vertexes according to their anti-clockwise distance from $A$, i.e., $A$ has number 0, and all the points on the convex hull have indices $0, 1, 2, ..., c-1$ in anti-clockwise order, where $c \leq n$ is the number of points on the convex hull. Now consider other two vertices $B, C$ such that $A < B < C$. We start with $A = 0, B = 1, C = 2$. Note that we have the following properties:

(i) for any two vertices $B_0, B_1$ such that $B_0 < B_1$, let $C_0, C_1$ be the corresponding vertex that achieve the maximum area respectively, then $C_0 \leq C_1$. Indeed, as the figure shows, the point $C_1$ must lie on or to the right of the dotted green line (parallel to the segment $A, B_1$). But then such a point must lie at or after $C_0$, since all other points between $B$ and $C_0$ are actually closer to the segment $A, B_1$ than $C_0$ is. (This is the same argument as in the recitation.)
(ii) For the pair \( A, B \), the maximum-achieving point \( C \) is any point such that, 
\[
\text{area}_{AB(C-1)} \leq \text{area}_{ABC} \geq \text{area}_{AB(C+1)}.
\]
This is clearly true due to the property of convex hull.

Back to our algorithm: when we find the maximum for a certain \( B \), we keep track of the corresponding \( C \) for which the area achieves the maximum. In the next iteration with the point \( B + 1 \), our search for maximum can start from this recorded point. Hence, for some \( B \) if we start with \( C = k \) and end with \( C = k' \), then we spend \( O((k' - k) + 1) \) time for this choice of \( B \). And by the monotonicity properties above, these \( k' - k \) values (over different choices of \( B \)) sum to at most \( c \leq n \). Hence the total runtime for a fixed \( A \) is \( O(c) = O(n) \). So the total complexity (over all \( c \leq n \) choices of \( A \)) is \( O(n^2) \).

(25 pts) 4. **Square.** Write a program that takes as input \( n \) (not necessarily distinct) points \((x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\), along with a side length \( s \) of a square. All these values are integers. Your program should output the location of the lower-left corner of an axis-aligned square of side length \( s \) which contains the largest number of the given points. (A square includes all the points on its boundary.) It should also output the number of points contained in that square. The time limit is 10 seconds (60 seconds for Python).

**Input:**

The first line contains two space-separated integers \( s \) and \( n \). \( 0 \leq s \leq 10^6 \), \( 1 \leq n \leq 5 \times 10^5 \). The following \( n \) lines each contain a pair of integers, which are the \( x \) and \( y \) coordinates of one of the points. \( 0 \leq x, y \leq 10^6 \).

**Output:** The output consists of two lines. The first tells the location of the lower-left corner of a square which contains the most points. This is a pair of integers (\( x \) first then \( y \)). The next line tells how many of the input points that square contains. In case there are several equally good solutions, output any of them. See the examples below for more details.

**Input1:**

\[
\begin{array}{c}
2 \ 7 \\
0 \ 0 \\
2 \ 0 \\
4 \ 0 \\
1 \ 1 \\
0 \ 2 \\
2 \ 2 \\
4 \ 2 \\
\end{array}
\]

**Output1:**

\[
\text{square lower-left corner} = (0, 0) \\
\text{count} = 5
\]

**Input2:**
(1 pt bonus) 5. **Extending a Classic On-Line Algorithm.** A classic homework problem in on-line algorithms is as follows: There’s a graph of two vertices (1 and 2) connected by an edge. A McGuffin is located at one of the vertices. There is a sequence of requests from vertices 1 and 2 to access the McGuffin. For one request, if it is from vertex \( i \) and the McGuffin is at vertex \( i \) the cost is 0. If the request is from vertex \( i \) and the McGuffin is at vertex \( 3 - i \), then the cost is 1. After any request the McGuffin can be moved to the other vertex at an integral cost of \( d \geq 1 \). An on-line algorithm for this problem must make this decision without knowing the future requests.

The problem is to give an optimally competitive algorithm to decide when to move the McGuffin to the other vertex. It turns out that the optimal competitive factor is 3. Here’s the algorithm that achieves this. Maintain a counter \( c \), which is the number of costly requests that occurred since the McGuffin was last moved. When this counter reaches \( 2d \), move the McGuffin, and reset the counter.

For example, suppose the sequence consisted of the following repeated pattern: \( 2d \) 1s, followed by \( 2d \) 0s. The cost of each cycle for the above algorithm is \( 6d \), whereas the cost of the algorithm that never moves is \( 2d \).

So your first task is to prove that the above algorithm is 3-competitive.

A natural extension to this problem is what happens on other graphs with more than two vertices. So specifically what happens to a complete graph on \( n \) vertices? The rules are the same. Accessing the McGuffin remotely costs 1. Moving the McGuffin costs \( d \). Design a 3-competitive algorithm for this more general problem.