Problem A. Aircraft Seats Arrangement

There is a total of $O(r)$ seats in the plane. Each assignment of a seat can be done in $O(r)$ time according to the rules described in the statement. So total time is $O(rn)$.

Problem B. Border on Map

Let's fix the direction of the line that splits the first polygon in half. If we find the two tangents to the first polygon parallel to our direction, we'll get two lines with that direction such that for one of them the polygon is entirely on the left side and for the other - on the right side. If we move the position of the line from one tangent to the other, the area of the first polygon that is on the left side of the line changes continuously, so we can find the line that splits the first polygon in half in that direction by binary searching between two tangents (maintaining that for one of the borders no more than half of the area is on the left side and for the other - no less than half).

If we choose a direction parallel to the y-axis in the same direction as the axis, then none of the second polygon's area will be on the left side of the line, that splits the first polygon in half, and if we choose the opposite direction, then all of it will be on the left side. The area that is on the left side changes continuously, so we can binary search again to find a direction that splits the second polygon in half as well.

If we calculate the area of the polygon on one side of the line by finding the intersections with the line in linear time and then finding the area, we'll get a solution in $\mathcal{O}(n \log^2(maxX \times precision))$.

Problem C. Conducting An Experiment

After the movement number i there is a total of k possible rat positions — along one side of the box. Right before the movement number i there are k possible rat positions, corresponding to those k positions. We will calculate $dp[i][j]$ — the minimum number of steps required to reach the j-th of these possible rat positions after the first i box movements. We can calculate it as $dp[i][j] = \min_{l=1}^{k}\{dp[i-1][l] + \text{dist}_i[l][j] + 1\}$, where $\text{dist}_i[l][j]$ is the distance between the square number l (after the box step $i-1$) and a square number j (right before the box step number i) inside a $k \times k$ subgrid of the grid.

There are $O(n^2)$ subgrids of size $k \times k$ and in each of them there are $4k-4$ squares, which might be used in some dist_i. We will precalculate the distances for each $k \times k$ subgrid and each of the $4k-4$ starts to the other squares. It can be done via bfs in $O(k^2)$ time for one $k \times k$ subgrid and one start, and for all of them it can be done in $O(n^2k^3)$ time.

After the precalculations, dynamic programming can be calculated in $O(qk^2)$ time. The total time is $O(n^2k^3 + qk^2)$.

Problem D. Divide And Conspire

Let's solve for each brother separately. Let $dp_{l,r}$ be the money that brother would earn if there remains a subsegment of the sequence from l-th to r-th NFT. Then depending on remainder of $r-l$ modulo m we'll know, whose turn it is, and set $dp_{l,r}$ to either minimum of $dp_{l+1,r}$ and $dp_{l,r-1}$ or maximum of $dp_{l+1,r} + a_l$ and $dp_{l,r-1} + a_r$. That works in n^2 time for each brother.

Actually we can find only those dp values where our brother makes his move. Then $dp_{l,r} = \max(a_l + \min_{l+1 \leqslant i \leqslant l+m} dp_{i,i+r-l-m}, a_r + \min_{l \leqslant i \leqslant l+m-1} dp_{i,i+r-l-m}$. We need to find only those dp values that have a specific residue of $r-l$ modulo m and we can find them in orders of increasing $r-l$ by layers. The minimums required to calculate the formula are sliding window minimums, so they can be found in $\mathcal{O}(n)$ for each layer. There are n/m layers in total and m brothers we need to solve the problem for. So the final solution works in $\mathcal{O}(n^2)$ with linear memory (because we only store the relevant layers of dp).

Problem E. Evacuation of Martian Rabbits

To minimize the number of jumps, each jump needs to go as high as possible, so in the highest vertex with a platform that is at most i vertices higher than the current one (for i-th rabbit). In two consecutive jumps i-th rabbit will go up by at least $i+1$ vertices, otherwise it could've made one jump with the same length. So until a rabbit escapes or can no longer make a jump, it will make at most $\frac{2n}{i}$ jumps. We can find, where a the highest possible jump will land a rabbit using HLD for the first 1 on a subsegment. Since all the queries for a single rabbit only cover the path from that rabbit to the root of the tree and each vertex is covered at most twice, so all the HLD queries will take $\mathcal{O}((\frac{n}{i} + \log n)\log n)$ time or $\mathcal{O}(n \log^2 n)$ time for all i.

Problem F. Fast Delivery

Let's look at the optimal path of the robot. It starts at position s, ends at f and only turns at integer coordinates, so every unit segment is fully traversed in one or both directions. If a unit segment is covered by some task then the robot has to traverse it at least once in the same direction as the task. All unit segments that are not between s and f are traversed either in both directions if the segment is not to the left of all tasks or to the right of all tasks or not traversed at all otherwise. If a unit segment between s and f is traversed in both directions then it is traversed at least 3 times in total since the robot starts on one side of it and finishes on the other. Now whatever the path has been previously we can change it to traverse the unit segments between s and f once if previously we traversed it in one direction and three times if we traversed it in both directions (we just need to turn around at the end of a contiguous segment that was traversed in both directions, go back to it's start and turn around again). The new path can still complete the same tasks the old one could and it traverses each unit segment the least amount of time possible, so it's not worse than the previous path

If we fix s and f than for each unit segment we know, in which directions we need to traverse it to complete the tasks. So the entire path is determined by its endpoints. Now we can find the optimal path with linear dp with 4 states at each relevant coordinate, describing if s and f were before this coordinate. If one of s and f is before, then we need to add 3 if the segment is covered by a task going in the opposite direction to s to f and 1 if it isn't covered. If neither or both were before than we need to add 2. Since we first need to do a scanline to know, which segment is covered by tasks going in which direction, the solution works in $\mathcal{O}(n \log n)$ time.

Problem G. Great Desert

First let's show that the optimal path is a polygonal line with vertices at the river's vertices, that is monotone by y. If we have crossed a segment of the river once, we'll never need to return to it again, so building a bridge in some point of the segment is equivalent to removing this segment. After some segments have been removed we need to go from one point to the other without intersecting some segments - here it's clear that the optimal path is a polygonal line that turns only at the segments' endpoints.

Now we can try to do dp to find the least amount of money needed to build a road from the first city to i-th vertex and end on a specific side of the river. We need to update the values of dp for one vertex through all previous vertices. So to do that in n^2 time all we need to know, is how many segments of the river does every segment between 2 of it's vertices cross.

Let's fix the first end of the segment - s and sort all the vertices after it by angle of the vector from s to that vertex. Let i-th point have position p_i in that order. Than for each index j we need to find the number of indices k between i and j such that $p_k < p_j < p_{k+1}$ or $p_k > p_j > p_{k+1}$. To do that we can go in order of increasing j and add 1 on an interval between p_{j-1} and p_j. Then the value at p_j is the needed value. We can do this with a segment tree to get an $n^2 \log n$ solution.

Problem H. Hairband and Nails

We can look at the process in a backward direction. Then, the nails are being added to the board and we should maintain the area of the convex hull.

The simplest way to do it is to maintain the upper hull and the lower hull of the points and the sum of cross products of consecutive points in these hulls. We will describe how to maintain the upper hull; the lower hull can be maintained in a similar way.

The upper hull will be stored sorted by x coordinates of the points in a set. If a point that we are trying to add is under the upper hull already, we don't have to do anything. Otherwise, the point should be added to this set, and then some of its neighbors (by x coordinate) should be removed until the convexity is restored. All these operations work in $O(n \log n)$ amortized time.

The total time is $O(n \log n)$.

Problem I. Installation Disks

There are $\binom{n+3}{3}$ different ms-swap operations and only $n!$ possible permutations. Thus, using bfs, we can find the distance from the given permutation to all the other permutations in $O(n! \cdot n^4)$ time. To speed up this solution, we can run this bfs from the permutation and the sorted permutation at the same time until some vertex is reached from both the start permutation and the sorted permutation.

Problem J. Jumbo And Food

If there are no black cells, the solution is trivial. Otherwise, we will denote T as the expected number of steps Jumbo will take to reach the bottom-right corner from a randomly uniformly chosen black cell. Then the expected number of steps Jumbo will take to reach the bottom-right corner from a cell (x, y) is $\min(c_{x,y}, b_{x,y} + T)$, where $c_{x,y}$ is the distance from the cell (x, y) to the bottom-right corner moving only on cyan cells (except the cell (x, y), which can be black), and $b_{x,y}$ is the minimal number of steps required to reach a black cell from the cell (x, y) (if the cell (x, y) is already black, we have to reach it again or reach another black cell).

So, $T = \frac{1}{k} \sum_{i=1}^{k} \min(c_{x_i,y_i}, b_{x_i,y_i} + T)$, where $(x_1, y_1), ..., (x_k, y_k)$ are the coordinates of all black cells. It can be shown that there is only one positive root of this equation. The function on the right is piecewise-linear and there are at most $k + 1$ intervals with different coefficients (since they change in points $c_{x_i,y_i} - b_{x_i,y_i}$). We can calculate the segments and coefficients on each segment and check whether the solution of a linear equation lies inside the corresponding segment. After solving the equation, the answer to the problem is $\min(c_{1,1}, b_{1,1} + T)$. This can be done in $O(k \log k)$ time.

We can calculate $c_{x,y}$ via bfs from the bottom-right corner. It can be seen that the values $b_{x,y}$ are used only for black cells and the upper-left corner, so we can calculate $b_{1,1}$ using bfs and for a black cell $b_{x,y}$ is either 1 (if it has a black neighbor) or 2 (otherwise).

The total time is $O(rc \log(rc))$.

Problem K. King Byteazar's Road

For each of the m possible starting points, we will calculate the number of required excavators. We will iterate through all the potholes and we will take each excavator into account when considering the first pothole it covers.

Let x be the coordinate of the pothole and y be the coordinate of the previous pothole ($y < x$). If the starting point is s, x is the first pothole filled by some excavator if and only if there exists k such that $y < s + km \leq x$. Therefore, we need to add 1 to the number of excavators with starting point s for all s such that there is a number in $(y, x]$ with the same remainder as s modulo m. So, we should add 1 to a (cyclic) subsegment of $1...m$.

It can be done with some data structure like a segment tree, or even in linear time, since we are interested only in the final values. The total time is $O(n + m)$ or $O(m + n \log m)$.

Problem L. Looking For Palindromic Path

At first we will describe the solution which works in time $O(\sum_{c \in S} e_c^2 + n^2 + q)$ where S is the set of lowercase English letters, and e_c is the number of edges with the symbol c.

We will call an unordered pair of vertices (v, u) *good* if there is a palindromic path from v to u. The pair (v, u) is good if one of the following holds:

- $v = u$;
- v and u are directly connected by an edge;
- There is an edge (v, x) with symbol c, and an edge (u, y) with symbol c and the pair (x, y) is good.

Then for a static graph, all the good pairs of vertices can be found with the following bfs-like algorithm:

1. All the pairs (v, v) are marked good and inserted into a queue Q; for each edge (u, v), the pair (u, v) is marked good and inserted into a queue Q.
2. Each step extracts a good pair (u, v) from the queue Q and runs through all the symbols c, through all the edges (x, u) with that symbol c and all the edges (y, v) with that symbol c. If the pair (x, y) is not yet marked good, it is marked and added to the queue Q.

It can be seen that this algorithm marks all the good pairs of vertices. In this problem, the edges are being added to the graph, so when the edge (u, v) with symbol c is added, we will run through all the edges (x, y) with symbol c, and if the pair (x, u) is marked good and (y, v) is not, mark the pair (y, v) and add it to the queue (and we should do the same for (x, v) and (y, u) and vice versa). Then we will continue running the loop described in the algorithm above.

To solve the problem, we will show that many of the added edges are useless and that there are at most $O(n)$ useful edges with each letter. For symbol c, among the edges with this letter, useful are only the edges which change the connectivity of components or the bipartiteness of some component.

Let edge (u, v) be useless. It means that there is a path between u and v of length $2k + 1$ which goes only through edges with letter c. Then each appearance of edge (u, v) in a palindromic path has some edge (x, y) with letter c in a symmetric position. We will replace this one edge with a path of length $2k + 1$, and the edge (x, y) we will replace with the path $x - y - x - ... - x$ of length $2k + 1$, thus the path is still palindromic, but has fewer occurrences of edge (u, v). We can repeat this process until there are no edges (u, v) left. So, it can be seen that the edge (u, v) is not necessary for a good pair of vertices and can be ignored.

The number of useful edges with letter c is at most $2n - 1$, and the first described solution then runs in time $O(|S| \cdot n^2 + q)$.

Problem M. Median On Path

Let's build a new bipartite graph from our graph. In the left part will be the vertices of the original graph, and in the right part — biconnected components of the original graph. Vertices are connected by an edge iff the vertex of the original graph belongs to the component. Note that the new graph is also acyclic.

A simple path from u to v can pass through vertex w iff it needs to visit some vertices from some biconnected component with w and either start and end in different vertices or at w. So in the new graph the only simple path between u and v needs to either pass through w or through some component that contains w. So $S(u, v)$ is exactly all vertices that either lie on the simple path from u to v or are adjacent to a vertex that lies on the simple path. We'll say that if a vertex corresponds to a component in the original graph, than there is no number on it, so those vertices don't affect the median.

Now the task is to find a median on a path and all of its adjacent vertices in a tree. Let's choose a root in the tree and move all values from vertices to their parents in a tree. Now the values on a path and all its neighbours is the values that were written in the root of the path and its parent before we moved the values and all the values on the path after the move. So now multiple values can be in the same vertex, but we need to find the n-th element on a path. To do that we can first traverse the tree, maintaining the count of each value on the path to the root in a persistent segment tree that counts the sum on a segment. And to answer the queries we need to simultaneously traverse the states of the segment tree in the endpoints of the path and their lca, similarly to finding n-th element on a segment in an array. Everything works in $\mathcal{O}((n + m + q) \log n)$.