Central European Olympiad in Informatics 2026
Day 1 - solutions

Birdwatchers

We can represent the problem with a tree of officers with the nodes weighted by their number of members. Note that if there are two senior officers (with influence $\geq M/2$), one must be the ancestor of the other. We are looking for the deepest node in a tree that has a subtree of size at least half the entire tree (taking into account the weights of individual nodes). This is essentially the centroid of a tree with a tie-break criterion that we want the centroid that is furthest from the root. We can start in the root and move down into the largest child as long as possible. However, the tree also undergoes some changes, where we can detach a subtree with root x that is attached to node y and reattach it somewhere else as a child of node z.

subtask 1 To perform the descent towards the centroid, we need to maintain a list of children for each node. When we move a subtree, we can update the two affected lists of children in $O(n)$. Then we perform a descent that requires $O(n)$ steps and compute the size of the subtrees on each step also in $O(n)$. This lets us recompute the solution in $O(n^2)$.

subtask 2 The bottleneck of the previous solution was the computation of subtree sizes. Instead, we can update only the subtree sizes of the ancestors at the locations of detachment and reattachment. This computes the answer in $O(n)$.

subtask 3 To improve the solution, we need a faster way of moving around the tree while still maintaining the ability of subtree relocations. We can do this using a sqrt-decomposition. We will form regions (subtrees) by processing the tree from leaves towards the root and keep forming a region until it exceeds $\sqrt{n}$ nodes. This will result in $O(\sqrt{n})$ regions of depth $O(\sqrt{n})$ but individual regions can have as many as $O(n)$ nodes (for example, a root node with many small children that are not regions themselves).

We haven't considered what happens when we relocate a subtree. We can split-off this subtree and let if form a new region that we attach somewhere else. If x was not already a root of some region and we had to split it off, its new region contains at most $O(\sqrt{n})$ nodes. We can search all of them to find the subregions that must be relinked to point to the newly formed region. To manage the growing number of regions we can rebuild the entire region structure of the tree in $O(n)$ after $O(\sqrt{n})$ relocations to achieve an amortized time complexity of $O(\sqrt{n})$.

We haven't described how to search for the centroid in this new structure. Descending from the root is problematic because we would need to update subtree sizes of all $O(n)$ ancestors, which we can't afford. Instead, we can move up the tree from y and z and on each of these two paths look for the nearest node that has a sufficiently large subtree. Once we reach the lowest common ancestor, the subtree sizes won't change anymore, which means that the centroid also hasn't changed from the previous update. This ascent can be performed by first moving over entire regions (and recomputing region sizes on relocations) and then over nodes within the last region. The height of this region is $O(\sqrt{n})$ and we can update the subtree sizes of individual nodes in the region only for the nodes that we actually need. This lets us compute the new centroid or determine that it hasn't changed in $O(\sqrt{n})$.

subtask 4 To be even more efficient, we will need a different representation of the tree. An Euler tour of the tree gives us a list of nodes, where subtrees correspond to contiguous sublists. If we represent this Euler tour with a balanced tree structure (such as a treap or a splay tree), we can efficiently delete and insert parts of this list, which corresponds to moving the subtrees. This representation is also known as Euler

Tour Trees. It simplifies moving the subtrees but complicates other operations such as finding ancestors of a node.

We will need to store some additional information. In the Euler tour we will also store the depths of nodes $d(x)$ in the original tree and in the treap that we built on top of the Euler tour, we will store the minimum depth of a node in the subtree. When we move a subtree rooted at x, the depths of all the nodes in the subtree of x change by the difference in depths of attachment nodes y and z. We can update this depth change of an entire subtree of nodes lazily by marking the change in the treap.

As we have already noted in the previous subtask, the new centroid will be located on the path between y and z or it won't change at all (the size of subtrees for ancestors of $\text{lca}(y, z)$ won't change and wouldn't affect the centroid search). We will lift each node y and z to find its lowest ancestor that has a sufficiently large subtree. We can do this with a kind of binary search, where we would consider 2^i-th ancestors by decreasing powers of 2. To do this we need to efficiently compute the k-th ancestor of a given node. If we consider the Euler tour, the k-th ancestor of node x corresponds to the rightmost item in the list with depth equal to $d(x) - k$. In the treap representation of the Euler tour, we can use the stored minimum depths to guide the search towards the rightmost node with appropriate depth.

The time complexity of this solution is $O((\log n)^2)$, because we need to consider $O(\log n)$ ancestors and determining the ancestor requires a $O(\log n)$ search in the treap.

A more advanced data structure that supports tree/forest modifications is the Link-cut tree. It can be used in a similar manner to solve the problem with an $O(\log n)$ amortized time complexity, but it wasn't required to score full points.

DFS

We are given an output of a depth-first search traversal of a graph with node ids and their depths. From this we can reconstruct the DFS tree structure with the help of a stack — we pop nodes from the stack until we reach an ancestor on the stack that has depth by one lower than the current node, which we then add to the top of the stack.

All of these tree edges must be present in the graph, but there could also be some additional edges. Cross-edges between nodes of different subtrees can't be present as they would change the structure of the DFS tree. However, back-edges from a node to some of its ancestors could exist. Note that such back-edges must link node x to some ancestor y such that x is larger than all of y's children. Otherwise, the output would be different because the DFS traversal would enter x before some of y's children. Because the children are visited in ordered by increasing numbers it is sufficient that x is larger than the last child z of y. In other words, if z is smaller than x, we can attach x to the parent of z without changing the output. We will need an efficient way of keeping track of the ancestors of the current node x so that we can count the number of ancestors smaller than some threshold.

The added edges are independent of each other, therefore the number of graphs is equal to 2^k, where k is the number of edges we can add. The number of new edges k can be rather large, $k = O(n^2)$. To compute the power of 2, we can use exponentiation by squaring.

subtask 1 There are so few nodes that we can afford to run the DFS traversal on all possible graphs (subsets of edges in a complete graph) with exponential time complexity.

subtask 2 Consider all $O(n^2)$ possible edges that can be added to the DFS tree, determine whether it is a back-edge and count relevant ancestors in $O(n)$ for a solution with a time complexity $O(n^3)$.

subtask 3 Keep track of ancestors on a stack but search for relevant ancestors in $O(n)$ for an $O(n^2)$ solution.

subtask 4 The output is a line graph with ordered nodes, for which we can compute the number of solutions directly. We can link every node to any of its ancestors. The solution is therefore $2^{(n-1)(n-2)/2}$.

subtask 5 The output is a line graph but the nodes are not ordered. We need to keep the processed nodes in a balanced tree structure that allows us to efficiently add new nodes and count the number of entries smaller than the new node. We can do this with a segment tree, Fenwick tree or use the red-black

tree with tree order statistics from C++ policy based data structures. The solution has time complexity $O(n \log n)$.

subtask 6 In addition to the solution for the previous subtask, we need to keep track of ancestors, which we add and remove from the balanced tree data structure.

Treasure Hunt

In this task we are looking for $k \leq 3$ treasures on a very large grid with the help of queries (x, y) that tell us a subset of directions from (x, y) that would bring us to some nearest treasure along some shortest path. The main complication is that other treasures can mislead us when looking for some treasure.

subtask 1 Looking for a single treasure is simple. We perform one binary search to find the column and another to find the row. Actually, we can even do these two simultaneously.

subtask 2 We need to find a first treasure T with the other treasure present as a distraction. Simultaneous search for its row and column can throw us off, so we'll first determine the column and then the row. We can start at an arbitrary position and move horizontally depending on the first query (we'll assume right) with a binary search to locate the column with some treasure — we search for a boundary between a cell with a right pointer and the adjacent cell to the right without it (this one must contain the treasure in its column). Next, we use the same logic to move down or up within that column — if we search upwards, we need to find the boundary between a cell with an up pointer and the adjacent cell without (which must be one of the treasures).

To find the second treasure U, we will search separately in four possible regions (right, left, up, down) formed by the diagonals through the first treasure T. Suppose that we search from the first treasure to the right along the same row. Some cells will point to the left towards the first treasure until something changes, which means we are close to the second treasure. We can use binary jumping (by increasing lengths of powers of two) to determine the location of such change and pinpoint the exact location where a cell points up/down to the second treasure with another binary search. This determines the column and we can find the row with another binary search within that column.

Be careful with corner cases where treasures lie on the same row or both treasures lie in the opposite corners of some square.

subtask 3 We have already described how to find the first treasure T. The same strategy works with three treasures.

When searching for the second treasure, we could face several treasures in the same region (for example, right). Previously, we were searching from T within the same row for some column that contains the second one. Now, we would like to be more precise and find the closest one. Let B be the closest cell to the right of the first treasure that does not point exclusively to the left. We know that one of the other two treasures lies somewhere on the diamond centered at B. We can binary search within the row to the right of B to find some column with the second treasure and then do the same within the column up or down.

Suppose we found the second treasure at point C, which lies on the diamond, where we expect it. We can start searching for the third treasure from C in three regions (up, down, right) in a similar manner. Otherwise, we know a line segment, where the third point should lie, but we found some treasure at a further point D. We can perform a similar search from point D in two directions because the line segment (side of the diamond defined by point B) intersects at most two regions of D. The location of the final treasure can be determined from the intersection of two line segments.

We need to be careful with situations where T and U lie on the same diagonal. In this case, you could find one from the other and vice-versa without ever finding the third one.

This gives us a solution with at most $17 \log n$ queries that we can further improve to at most $11 \log n$. For example, by taking into account certain orientations of the first two found treasures, we can save on the number of binary searches that are required to locate the last treasure. One such optimization is to consider projections of the second found treasure to the borders of the grid. If we query these points, we can quickly determine in which region the third treasure lies or that it's within the same row or column as U.