Problem A. Bipartite Matching

Just use the algorithm described at the lecture. Any $O(VE)$ algorithm will do.

Problem B. Minimal Vertex Cover

By Kőnig's theorem the number of vertices in the minimum set cover is the same as the number of edges in the maximum matching.

To find the set itself, run depth first search from all vertices of the left part, not covered by the given matching. Not visited vertices of the first part and visited vertices of the second part form the desired minimum set cover.

In order for the complexity to be $O(E)$, not $O(VE)$ it is important to keep `visited` marks between DFS runs, do not make new DFS calls from vertices already visited.

Problem C. Path Cover

Create a bipartite graph H, for each vertex u of the original graph create two vertices u_1 and u_2 in both parts. If there is an edge uv in the original graph, add an edge u_1v_2 to the graph H.

Now find the maximum matching in H. Consider edges of the maximum matching in H in the original graph G. They form the set such that for each vertex there is at most one edge starting in this vertex and at most one edge ending in this vertex. Such set of edges generally consists of paths and cycles, but since the graph is acyclic, in only consists of paths.

Consider vertices of G. Initially there are n vertices, let them correspond to n paths. Add edges of the matching one by one, each time you add an edge you reduce the number of paths by one. So if the maximum matching contains k edges, the answer is $n - k$.

Problem D. Cubes

Create a bipartite graph. The vertices of the first part letters of the sister's name, the vertices of the second part are cubes. Connect the letter to the cube if the cube contains this letter.

Now Senya can compose the name if there is a matching that covers all the vertices of the first part. Find the maximum matching and check if it does.

Problem E. Domino Tiling

If $2b \le a$ it is always cheaper to use squares for tiling, so the answer is bf where f is the number of free cells. Otherwise we must try to use as many domino tiles as possible.

Paint the grid as a checkerboard, the cell (x, y) will be white if $(x+y) \bmod 2 = 0$, and black otherwise. Now create the bipartite graph, the vertices of the first part correspond to free white cells, and the vertices of the second part to free black ones. Connect two cells if they are adjacent. Now find the maximum matching in the resulting graph, the number of edges in it is the maximum number of domino tiles that can be used. If the size of the matching is m the answer is $ma + (f - 2m)b$.

Problem F. Birthday

There are three steps from the maximum matching in the graph to the maximum bipartite clique required in this problem.

The size of the maximum matching is equal to the size of the minimal vertex cover.

The complement of any vertex cover is an independent set (no two vertices connected by an edge).

The independent set in the inverse of the graph is the clique in the original graph.

So we invert the graph, find the maximal matching, find the minimum vertex cover, its complement is the maximum independent set which is the maximum bipartite clique in the original graph.

Problem G. Relocation

First let us make binary search of d. Now we want to check if the answer is at most d.

Note that being able to communicate is transitive and symmetric, so it is enough to make sure that there is a spy u such that it can communicate with any other spy. Let the city u contain a spy after relocation. Make u the root of the tree. Since the spy at u must be able to communicate with any other spy, for any other city v with a spy all cities between u and v must contain a spy.

First move each spy as far as possible towards the city u, mark the city that it comes to. All marked cities will either contain a spy after relocation, or will be on a path from u to a city that contains a spy. So it must contain a spy any way. Now all cities that are on a path from a marked city to u must contain a spy. And we can always move spies so that no other city contains a spy.

Create a bipartite graph. Vertices of one part are the cities that must contain a spy, the other part are spies. Connect a city to a spy if the spy the distance between the city and the initial location of the spy doesn't exceed d. The answer to the original problem is at most d if for some u the constructed graph contains a matching that covers all vertices of the first part.

Time complexity: binary search $O(\log Max)$, iterate over n possible variants for u, creating a graph is one DFS with $O(n)$ complexity, matching is at most $O(n^3)$. The total complexity is $O(n^4 \log Max)$.

Problem H. Postfix RLE

Create a parse tree of the expression, the internal nodes are the operations, the leafs are the variables.

If some internal node u with operation "$\circ$" has a child v with the same operation "$\circ$" we can remove the node v, moving its children to become the children of u. Do it until no node has a child with the same operation. Now the tree is not binary, each internal node can have 2 or more children.

Now we can run dynamic programming. Let $dp[u][c]$ be the minimal number of blocks in representation of u subtree that starts at a character c. The optimal way to create an expression for u is to position its children's expressions in some order followed by $\circ \circ \ldots \circ$.

If there are two children v and w, positioning their expressions e_v and e_w one after another we get an expression $e_v e_w$. If a child node v is an internal node, its postfix expression ends with its operation, so the block size of the resulting concatenation is the sum of blocks sizes e_v and e_w. So the only freedom we have is to position expressions for children that are leafs before expressions for children that are internal nodes. Each such positioning reduces the total block size by 1.

It is easy to see that $dp[u][c]$ differ by at most 1 for all c. Let us consider those values that are optimal for the given u, denote the of such c for u as $Opt(u)$. The only reason to use $dp[u][c]$ for a character c not from $Opt(u)$ is to force it to come forward to merge some single-letter expressions to it. However, this increases block size by 1 and then reduces it by 1 again, so it is never optimal to do so. Therefore we should only start expressions with characters from $Opt(u)$.

Now for each u create a bipartite graph. The first part consists of characters that are in at least one child leaf of u. The second part consists of internal nodes v that are children of u, connect c to v if $c \in Opt(v)$.

To find the $dp[u][c]$ value we find the maximum matching in the given graph, this is the number of blocks we can save. Any character in the leaf child of u is in $Opt(u)$. Also for each internal child node v of u: if c is in $Opt(v)$ and there is a matching that doesn't cover v, then c is in $Opt(u)$.

Problem I. Graph Game

This game is called Undirected Vertex Generalized Geography. It can be solved in polynomial time: the vertex u is winning for the first player, if and only if every maximum matching covers u.

Proof. Let there be a maximum matching M that doesn't cover u. Then the winning strategy for the second player is the following. No matter, how the first player moves, the token comes to a vertex v covered by M, since M is maximum and the edge uv cannot be added to it. Now the second player always moves along the edge from M. The first player can never move to a vertex not covered by M since the path of the token would then be an augmenting path. So the second player always has a move and eventually wins.

Let every maximum matching cover u. The first player fixes any maximum matching M and moves along its edges during his moves. The second player cannot move to a vertex not covered by M since otherwise the path of the token would allow to change M to M' that doesn't cover u. So the first player always has a move and eventually wins.

To find the set of vertices not covered by some maximum matching, first find some maximum matching M. Now for each vertex u covered by M, in turn, remove it, and try to find the augmenting path. If it exists, there is a maximum matching that doesn't cover it.

The total complexity is $O(VE)$.

Problem J. Drawing Windows

The key idea is that we must remove all concave vertices. To do so, we can draw a segment from each of them. Each such segment splits the polygon into two parts, that totally have one less concave vertices. When there are no concave vertices, all parts are rectangles.

To minimize the number of parts we would like to remove two concave vertices at a time. For each pair of concave vertices that can be connected by a horizontal or vertical segment, that doesn't intersect the border of the polygon, let us draw such segment. We would now like to choose the maximum number of segments such that no two segments intersect. If we create a graph with these segments as vertices, and edges connecting intersecting segments, the problem reduces to finding the maximum independent set in this graph. And since it it bipartite, the answer can be found using maximum matching algorithm.