Editorial

Tasks, test data and solutions were prepared by: Dominik Fistrić, Bartol Markovinović, Bojan Štetić, Paula Vidas, Pavel Kliska, i Krešimir Nežmah. Implementation examples are given in the attached source code files.

Task Kaučuk

Prepared by: Pavel Kliska
Necessary skills: for-loops, if, string equality

For the subtask in which $n \leq 3$, it is sufficient to examine a few possible cases. Because only the sections can stand by themselves, the first command has to be `section`. The second command can be either `section` of `subsection`. Depending on the second command, the third is either `section` or `subsection`, or in the other case `section`, `subsection` or `subsubsection`. These five options can be checked using if statements.

In the subtask in which only the `section` command appears, it is sufficient to number the titles of sections with positive integers from 1 to n.

The subtask in which only the `section` and `subsection` commands appear are solved analogously to the entire solution, but without taking into account the `subsubsection` commands.

For all the points, the task can be solved by keeping track of a counter for each level: S_0, S_1 and S_2. In the beginning all of the counters are set to zero. We iterate with a for loop over the input:

- if the current command is `section title`, we increase S_0 by one and print S_0 `title`, and set S_1 and S_2 to zero because in each section the numeration of the (sub)subsections starts over again
- if the current command is `subsection title`, we increase S_1 by one and print $S_0.S_1$ `title`, and set S_2 to zero because in each subsection the numeration of the subsubsections starts over again
- If the current command is `subsubsection title`, we increase only S_2 and print $S_0.S_1.S_2$ `title`.

Task Kutije

Prepared by: Paula Vidas and Krešimir Nežmah
Necessary skills: DFS or BFS, permutations

For the second subtask, it is sufficient to go through all of the meetups and pairs of meetups, and for each possibility determine in which box the toy in question will end up in and check if this is the desired box.

In the third subtask, for each question we can recursively determine in which boxes the staring toy can end up in. If we are currently at some box, we can go through all friends, check where would the toy end up after a meeting with that friend and then make a recursive call in case we haven't already visited that box. In the end, we check if the toy has ever visited the desired box.

For the first subtask, for each toy we can do the following: we determine in which box it ends up in after one meetup, then after two meetups, then three, and so on, until we return to the starting box. Then we know that the boxes in the meetups will repeat periodically after that. So, it is sufficient to store for each toy all of the boxes it can end up in, and then we can answer the questions.

For the whole solution, the problem can be modeled by a (directed) graph. The nodes are labeled $1, \ldots, n$. For each meetup and every $i = 1, \ldots, n$, we add an edge from the node p_i to node i. Notice that the movement of a toy during a sequence of meetups is actually a path in this graph, and that every path also corresponds to a sequence of meetups. Also, in this graph it is true that if there is a path from node x to node y, then there is path from y to x. Thus, the answer to the question is affirmative if and only if the

nodes a and b are in the same connected component of the graph. The connected components can be found using BFS or DFS.

Task Hiperkocka

Prepared by: Paula Vidas and Krešimir Nežmah
Necessary skills: ad hoc

A single tree can be placed in the following way: root the tree in an arbitrary node, and place that node on an arbitrary node of the hipercube (say 0). Then we do a DFS on the tree, and when moving from a tree node that is placed on the hipercube node x to a new node using the i-th edge, we place it on the hipercube node $x \oplus 2^i$.

The rest of the trees can be placed as follows: for each $x \in \{0, \ldots, 2^n - 1\}$ that has an even number of ones in binary, we take the hipercube nodes on which the first tree was placed and *xor* their labels with x. Notice that in such a way we obtain 2^{n-1} trees.

A proof that the described trees make a tiling is left to the reader.

Task Magneti

Prepared by: Bartol Markovinović
Necessary skills: combinatorics, dynamic programming

Define the length of a permutation of magnets as the least number of adjacent slots used for placing the magnets in that order without attracting each other. Say that the order of the magnets is fixed and that the length of the permutation is d. If the length of the permutation is greater than l, then it's impossible to place the magnets on the board in that order. If the length of the permutation is less than or equal to l, then what's left is to arrange the remaining $l - d$ empty slots among the magnets so that there are a total of l slots on the board. Each of these $l - d$ remaining empty slots can be arranged among $n + 1$ "gaps" in the array, that is either before all magnets or after all magnets or between any two adjacent magnets. The number of ways to arrange these empty slots can be calculated with a combinatorial trick called Stars and bars and it equals to $\binom{l-d+n}{n}$.

For the first subtask, the length of each permutation of magnets is $(n-1) \cdot r + 1$ (if r denotes the radius of activity of the magnets), so the answer is $n! \cdot \binom{l-(n-1)\cdot r-1+n}{n}$. The binomial coefficients can be precomputed using Pascal's triangle.

For the second subtask, it's possible to go over each permutation of the magnets, for each one calculate it's length and the the number of ways to arrange the remaining slots. The total complexity of this is $O(n! \cdot n)$.

For the entire solution, we use dynamic programming to calculate for each d between 1 and l how many permutations have length d, and then multiply this number by $\binom{l-d+n}{n}$.

We sort the magnets by increasing radius and build the permutation with the following dp:

$$dp[i][j][d] = \text{number of ways to arrange the first } i \text{ magnets in } j \text{ groups such that the sum}$$

$$\text{of the lengths of the groups is } d.$$

One group represents a segment of the permutation that is being built and is comprised of magnets and the least amount of empty space between them. The transition of the dp actually consists of adding a new magnet to one of the groups, which can be done in three ways:

- creating a new group that is made just from this magnet,
- adding a magnet to one of the ends of one of the already existing groups, or

- connecting two existing groups by placing the new magnet between them

The number of permutations which have length d will be stored in $dp[n][1][d]$ (all of the magnets are in a single group of length d). The time complexity of this solution is $O(n^2 \cdot l)$ because there are $n \cdot n \cdot l$ states and the transitions are calculated in $O(1)$. For more details, please see the official implementation.

Task Osumnjičeni

Prepared by: Krešimir Nežmah
Necessary skills: set, two pointers, binary lifting

The first observation is that for each question, the optimal sequence of lineups can be built by starting from the suspect p_i and adding the suspects one by one from left to right into the current lineup, until at some point this is not possible. When we reach such a suspect that can't be added, we start a new lineup and repeat this process until we reach the suspect q_i.

We can think of this in the following way: for each index x ($1 \leq x \leq n$), define $nxt(x)$, which will denote the smallest index greater than x such that the suspects from x to $nxt(x)$ don't form a valid lineup. In other words, if we start building a lineup from x, $nxt(x)$ is the first suspect we won't be able to add into our set, and he represents the start of a new lineup.

The answer to each query now reduces to finding out how many times must we iterate the function nxt, starting from p_i until we pass q_i, that is, so that $nxt(...(nxt(p_i))...) > q_i$. The solution of this problem has two parts: how do we efficiently calculate $nxt(x)$ for each index x, and then how to answer the queries efficiently.

$nxt(x)$ can be calculated with the "two pointers" idea. Notice that if $x < y$, then $nxt(x) < nxt(y)$ because when moving from index x to $x + 1$, the lineup we have built thus far remains valid even after we remove x from the lineup, and possibly it should be extended to the right. To make the insertions and deletions from the current lineup efficient, we have to use a data structure which can support these operations quickly. It is sufficient to use a set container from c++ STL, in which we'll store the left boundaries of the currently inserted height ranges. When inserting a new interval $[l, r]$, we find the leftmost left boundary that is to the right of l (using the lower_bound function) and check that it isn't in the interval $[l, r]$. Analogously, we can find the rightmost left boundary smaller than l and check that l isn't contained in that interval. If the mentioned conditions hold, we can add the current suspect to the lineup and add the left boundary to the set.

We can answer the queries using "binary lifting". We'll use $par[i][j]$ to store $nxt(...(nxt(i))...)$, where nxt is being applied 2^j times. Notice that for all indices i it holds that $par[i][0] = nxt(i)$, and also $par[i][j] = par[par[i][j-1]][j-1]$. Using the mentioned formulas, we can calculate $par[i][j]$ for all $i = 1, ..., n$, and $j = 0, ..., \lfloor \log n \rfloor$. We can now answer each query in $O(\log n)$ by descending over j and at each step moving to the 2^j-th ancestor if it does not pass the right boundary (q) of the current query.

The total time complexity of the presented algorithm is $O((n + q) \log n)$. The subtasks were ment for competitors who managed to figure out how to calculate nxt efficiently, or who knew how to answer queries quickly, but who didn't know both.