Editorial

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

Task Med

Prepared by: Pavel Kliska
Necessary skills:

This problem is intended to check whether participants can write their own comparator function for sorting. Assume that the sixth round was also held. A competitor with name x and points $b_{x1}, b_{x2} \ldots, b_{x6}$ will be ahead of competitor with name y and points $b_{y1}, b_{y2} \ldots, b_{y6}$ if and only if $b_{x1} + b_{x2} + \cdots + b_{x6} > b_{y1} + b_{y2} + \cdots + b_{y6}$, or if the sums are equal and x is lexicographically smaller than y.

For each competitor we can look at the worst and best possible scenario - all competitors except them win 600 points and they get 0 points, and vice versa. For each scenario we sort the ranking according to the comparator and find the best and worst possible positions for them.

In C++ this can be done by writing a custom comparator function and then using `std::sort`, while in Python this can be done by sorting twice with different key functions for `list.sort`.

Task Zemljište

Prepared by: Bartol Markovinović
Necessary skills: two pointer method, prefix sums

Without loss of generality assume that $a \leq b$.

Let's fix the first and last row of the plot of land to be chosen. Now the goal is to determine the best left and right end columns of the plot of land. Since the rows are already fixed, when choosing columns we'll take only the cells between the two end rows. Therefore, using prefix sums we can calculate for each column the sum of all cells in that column which are between the two rows. Then the problem becomes the following: given a sequence of numbers determine which interval has sum closest to the numbers a and b.

This problem can be solved using the two pointers method. We'll keep track of two pointers (indices in the array) l and r, which will point to the leftmost and rightmost end of the interval respectively. We'll move the r pointer to the right while the sum of the range is smaller than a, and for this sum we'll check if it's optimal. Then we'll move the pointer l one position to the right and again update r until the sum passes a. Each pointer will be moved a total of $O(n)$ times so the total time complexity of the algorithm (together with fixing the rows) is $O(n^3)$.

Task Naboj

Prepared by: Bartol Markovinović and Krešimir Nežmah
Necessary skills: graph theory, DFS, topological sort

The metal balls from the problem can be thought of as nodes of a graph, and the copper wires can be thought of as edges. When the professor charges a ball this now corresponds to directing all the edges either towards that node or away from it. The problem now boils down to checking whether it is possible to obtain the desired directed graph starting from a given undirected graph using such operations.

Let's look at what happens if the graph contains a directed cycle. Assume that there exists a solution, i.e. a sequence of moves which obtains the desired graph. Suppose d is the index of the last node in the cycle which has been charged. Then, all of the edges from the cycle connected to this node d would have to

point either towards d or away from it, which is impossible since there should be an edge both entering and exiting it. Therefore, if the final directed graph contains a cycle, then the answer is -1 and it is not possible to obtain the desired directions.

We can check whether there is a directed cycle using a DFS traversal of the graph. If the graph doesnt't contain a cycle, we can do a topological sort on the nodes. If we iterate over the nodes in this order, since there are no cycles, we can find a simple sequence of moves. Just charge the nodes in the order of the topological sort.

Task Palindromi

Prepared by: Krešimir Nežmah
Necessary skills: palindromic tree, merging sets small to large

For the first subtask, we can iterate over all substrings of the new string after each concatenation and check (e.g. using *hashing*) how many different palindromes there are. In this way, for a word of length l we can find the number of subpalindromes in $O(l^2)$ so the total time complexity is $O(n^3)$. We can notice that when connecting two strings, it is enough to iterate over only the substrings which begin in the first word and end in the second one. In this way, connecting two words of length a and b is done in $O(ab)$, which turns out to be $O(n^2)$ in total, solving the second subtask.

The following fact is useful for the remaining subtasks: if we append a character c to any string s to the left or to the right, the number of different palindromes of s will increase by at most 1. Consequently, the total number of distinct palindromes in a string of length n can be at most n. Furthermore, there exists a data structure which maintains all the different subpalindromes of a string s and the relations they have to each other. This data structure is called a palindromic tree or eertree. As mentioned, there are n subpalindromes and it's possible to maintain this data structute using $O(n)$ space and amortized $O(n)$ time.

In short, each subpalindrome of s is represented by a node. If p is a subpalindrome of s and c is a character such that cpc is also a subpalindrome of s, then there is an edge between p and cpc labeled with c. Additionally, for each subpalindrome p we'll have a *suffix link* towards the longest proper suffix of p which is also a palindrome. The eertree supports the operation `add`(c) which appends the character c to the end of the current string s. This function can be implemented in amortized $O(1)$ in the following way: at all times we maintain the longest suffix t of the current string s which is also a palindrome. When adding the character c, if there is another character c in front of t, we simply extend t to ctc. Otherwise, we iterate through the suffix link chain t, `suf`$[t]$, `suf`$[$`suf`$[t]]$, ... until we find the first palindrome in front of which is c. Implementing this idea (according to the link provided above) was suffiecent for the third subtask.

For the final subtask, we need to make a couple of modifications:

- Instead of the time complexity of `add`(c) being amortized $O(1)$, we can make it so that it is (non-amortized) $O(\sigma)$, where σ is the alphabet size. In our case, the alphabet consists only of the characters `0` and `1` so we consider the time complexity to be $O(1)$. Along with `suf`$[node]$, it is enough to keep track of two additional values: $\texttt{suf}_0[node]$ and $\texttt{suf}_1[node]$, which represent the longest proper suffix in front of which is the character `0` or `1` respectively. All the mentioned values can be computed in $O(1)$ when adding a new node.
- Instead of having a function `add`(c) which adds the character c to the right, we'll support adding character both to the left and to the right. To do this we maintain both the longest prefix and the longest suffix palindrome of the current string. When adding a character to the right, this effects only the longest suffix palindrome, and when adding it to the left it effects only the longest prefix palindrome. The only exception is when the entire string becomes a palindrome, in which case the longest prefix and suffix palindromes are equal and are equal to the entire string. Since the time complexity is no longer amortized, appending to both sides is done in $O(1)$.

- We'll append the smaller string to the larger string. If the right string is smaller than the left string, we successively add characters from the right string to the right end of the left string. If the left string is smaller, we successively add characters from the left string to the left end of the right string. It can be shown that the number of times a character gets added to a string is $O(n \log n)$, and since we hava a data structure which supports these additions in $O(1)$, the total time complexity is also $O(n \log n)$.

Task Superpozicija

Prepared by: Pavel Kliska and Krešimir Nežmah
Necessary skills: ad-hoc, greedy, prefix sums, segment tree or set

First, let's consider a way to determine if a sequence of parentheses is valid. In the given sequence, we replace each open parenthesis with `+1`, and `-1` for closed. Then we take the prefix sums over this sequence. It can easily be shown that a sequence of parentheses is valid if and only if every prefix sum value is ≥ 0 and the total sum equals 0.

In the first subtask, for each test case we can try out all 2^n possibilities. Since the worst case is when n is maximal and the number of test cases is $\frac{t}{n}$, the total time complexity is $O(2^n \frac{t}{n})$.

In the second subtask, if there are two open parentheses in pair, it is better choose the one on the left, and if there are two closed parentheses in a pair, it is better to chose the right one. The proof of this fact is a consequence of the characterization of valid sequences discussed in the first paragraph.

For the third subtask, we can imagine having n empty slots and for each slot having a choice between two parentheses. If the two parentheses in a pair are the same, then we don't really have a choice and we are free to choose any of the two for the corresponding slot. If the parentheses in a pair are different, we have a choice between placing an open or a closed parenthesis in this slot. Regarding the slots for which we have a choice, it's best to choose in such a way that there is a prefix of open parentheses and a suffix of closed parentheses. Exactly how many open/closed parentheses should there be is determined unambiguously from the fact that there should be an equal number of open and closed parentheses in total. In the end, we also check if the formed selection is valid.

For the whole solution, we can picture a choice of parentheses in the following way. We have a sequence of numbers 0, 1 or -1 of length $2n$. For each pair of parentheses, we discard one of them and place a 0 in its position, and we choose the other one and place a $+1$ or -1 depending on whether it is an open or a closed parenthesis. This leaves us with a sequence of n zeroes and n values ± 1 over which we take the prefix sums to check whether it's valid.

Similarly as in the second subtask, if two parentheses in a pair are the same, we choose the left one in case they are open, and the right one if they are closed. What remains are the pairs of type `()` i `)(`. Initially, for each such pair we'll choose the closed parenthesis, and then we have to decide for some of these pairs to switch our choice back to open. If a pair of parentheses is on positions a and b and if we decide to switch, let's consider what effect this has on the array of prefix sums. Regardless of whether we are dealing with a pair of type `()` or `)(`, the effect is that we add $+1$ on the suffix starting from a and $+1$ on the suffix starting from b. Therefore, we can represent each pair of parentheses as a range $[a, b]$, and some of these ranges ought to be *activated*, that is we should add $+1$ on the mentioned suffixes. Since there should be an equal number of open and closed parentheses in total, it is possible to precisely determine how many ranges should become activated.

What remains is to determine which ranges to activate. We claim that the choice should be done in the following manner: in the array of prefix sums we find the leftmost position with a negative value, call it c. It is clear that we should choose at least one range that will make this value nonnegative, so we should activate at least one range $[a, b]$ such that $a \leq c$. Out of all such ranges, it is optimal to choose the one which minimizes b. By repeating this procedure we obtain a solution, and if at some point there is no range which meets the conditions, the answer is `-1`. Implementing this process can be done with data

structures such as a segment tree, a set or by a sweep line on the array by storing information for suffix updates on future positions in the array.