Central European Olympiad in Informatics 2026
Practice - solutions

Beautiful Subsets

In this task we need to find the largest subset of nodes that are all of the same color and lie on some path in the tree (potentially along with some other nodes of a different color). We need to report the size of the largest subset for each of the colors in the tree.

subtask 1 The number of nodes is so low that we can afford to consider all possible subsets and check whether they lie on a path. There are $O(2^n)$ subsets and we can traverse the tree for a path from the lowest node in the subset to all others in the subset. If the path to the furthest one contains all others, we found one candidate. The time complexity is $O(n2^n)$.

subtask 2 Exponential solutions are too slow but the second subtask admits solutions with quadratic time complexity. We can solve the problem independently for every possible color in linear time for an overall time complexity $O(bn)$. Let us process the nodes from the leaves towards the root and consider the current node as highest node on the path. To maximize the number of colored nodes (of the current color) on the path, we would pick the two subtrees that contain the longest paths of colored nodes toward their leaves. We can compute the length of the longest colored path from the children's values.

subtask 3 To improve the solution, we could process colors simultaneously. The idea of keeping track of the longest paths of each color starting from the current node seems useful and we can store them in a map (we only store the longest paths to colors that are actually present in the subtree). To merge the results from children into the result of the current node, we can employ the small-to-large strategy of merging the results from smaller children into the largest one. This results in $O(n \log n)$ insertions with every insertion taking $O(\log b)$ time for an overall time complexity $O(n \log n \log b)$.

subtask 4 For our final approach, we will again process colors independently. However, we will compress the tree for each color c and retain only the nodes of color c (let there be n_c of them) and some other relevant nodes. The solution on the compressed tree remains the same: we consider two largest child paths for each node. We will compress the tree so that it is smaller but has the same result. The relevant nodes that we retain are those of color c as well as their parents. If some node and its parent both differ from the current color, we can delete the node in the compressed tree without changing the result. We can also delete the nodes that differ from c and don't contain any descendants of color c. This requires some preprocessing for efficiently compressing the tree with respect to each color but gives us a tree of size $O(n_c)$. Note that this does not maintain the lowest common ancestors in the compressed tree as is usually the case in constructing the virtual/auxiliary trees. All operations can be performed in linear time, which gives us a solution with time complexity $O(n + b)$.

Cave

We need to determine the number of cave formations (stalagmites, stalactites) that intersect with a horizontal line (railway) at all relevant heights.

subtask 1 The number of formations n and their heights v are both rather low. Therefore, we can simply afford to try every height and compare all formations against it in $O(nv)$.

subtask 2 The number of formations is larger but their heights are still relatively low. It is helpful to sort the stalagmites and stalactites by their sizes. Then, we can consider increasing heights of the railway and update the number of intersecting formations by moving along the sorted list of stalagmites and the sorted list of stalactites. This gives us a solution with time complexity $O(n \log n + v)$.

subtask 3 In this subtask, the heights are no longer conveniently limited. But we can adapt the previous solution. Instead of all possible integer heights, we need to consider only those that are present in the input as we know that the result doesn't change for heights in between.

Another approach is to maintain a list of changes. For each height we store by how much the number of overlapping formations changes. If we consider increasing heights, a stalagmite of size k_i implies a change of -1 at $k_i + 1$ and a stalactite a change of $+1$ at $v - k_i + 1$. Afterwards, we need a single sweep to add up the changes. A *map* structure is convenient for maintaining these values and iterating over them in an ordered manner.

Wagon Sorting

We need to sort the wagons using two queues by moving the wagons from the front of the queue to the back of the same or the other queue. First, we need to count them. We can do this by moving them from one queue into the other until the first one is empty.

The simplest method is to implement selection sort. One way is the following. We perform $n - 1$ iterations and on the i-th iteration we move a prefix of $i - 1$ already sorted wagons to the back and move the i-th wagon to the second queue. Then, we compare it with the remaining wagons from the first queue so that the smallest wagon remains in the second queue at the end. We cycle over the first queue again and insert the waiting wagon from the second queue at the i-th position. This increases the length of the sorted prefix of the first queue. Overall, it requires $O(n^2)$ operations.

A more efficient way is to implement some kind of merge sort. We will make $O(\log n)$ iterations and on every iteration we will double the length of sorted runs: we start with sorted runs of length 1, obtain sorted runs of length 2, then 4, etc. To merge the front two runs, we move the first one into the other queue. Then we perform a merge of the two runs and add them at the back of the first queue. We repeat this until we have merged all pairs of runs into twice longer runs. This requires approximately $3n \log n$ operations.

To make this more efficient, we can initially split the wagons into two queues. When we are merging the runs, we can put the merged run at the end of alternating queues. This way each queue contains some shorter runs followed by some longer runs. After we finish with the short runs, the longer ones are already distributed over the two queues ready for the next merge. This approach requires a bit more than $2n \log n$ operations.