Editorial

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

Task Autići

Prepared by: Krešimir Nežmah and Marin Kišić
Necessary skills: for-loop, ad-hoc

We'll show that it's always best to connect the garage with the minimum d_i with the remaining $n-1$ garages.

To connect all of them, we need at least $n-1$ roads (initially there is only one garage, and for each additional one we need one more road to connect it with the rest). In the optimal solution we will build exactly $n-1$ roads because it is never beneficial to connect two garages if there is already a path between them.

Therefore, the total length will be an expression consisting of $2(n-1)$ terms. Each garage needs to be connected to at least one road, so each of the numbers $d_1, d_2, \ldots, d_n$ must appear at least once in this expression. The smallest possible value of the remaining $n-2$ terms is precisely the value of the smallest d_i. Notice that this is actually achieved if we connect the garage with the smallest d_i with the rest.

Task Autobus

Prepared by: Paula Vidas
Necessary skills: dynamic programming

We model the problem as a directed graph.

The first two subtasks can be solved by iterating over all paths of length k, and storing the shortest distance between every pair of nodes. The time complexity is $O(n^k + q)$.

To fully solve the problem we use dynamic programming. Let $\mathrm{dp}[d][x][y]$ denote the length of the shortest path between x and y which uses at most d edges, and let $\mathrm{w}[x][y]$ be the length of the (shortest) edge between x and y, or ∞ if no such edge exists.

In the beginning, we initialize all the paths of length zero: $\mathrm{dp}[0][x][x] = 0$ and $\mathrm{dp}[0][x][y] = \infty$ for $x \neq y$. When making a transition from d to $d+1$, for each pair of nodes (x, y) we try to go from x to some node z using at most d edges, and from z to y using just one edge:

$$\mathrm{dp}[d+1][x][y] = \min_z \mathrm{dp}[d][x][z] + \mathrm{w}[z][y].$$

The complexity is $O(kn^3 + q)$, which solves the third subtask.

Notice that the shortest path between two nodes cannot use more than $n-1$ edges (otherwise we would visit some node twice, i.e. we would have a cycle which could be removed to obtain a shorter path). Therefore, if k is bigger than $n-1$, we can set $k = n-1$ and the answer will be the same.

Task Izbori

Prepared by: Bartol Markovinović and Krešimir Nežmah
Necessary skills: prefix sums, segment tree with propagation, amortised complexity

The problem requires us to count the number of intervals in the array A which contain a "dominant" element, that is, a number which appears more times than all the other numbers combined.

To solve the first subtask, it was sufficient to make a brute-force solution in $O(n^3)$ which iterates over all the intervals and checks if there is a dominant element. Solving the second subtask requires speeding this solution up to work in $O(n^2)$. One way to do this was to fix the left end of the interval, and while the right end is moving to the right we keep track of the number of elements in the segment and check to see if there is a dominant element.

To solve the problem entirely, for each value we'll count the number of segments in which this value is dominant. Fix some value x and make a new array B which consists of the numbers 1 and -1. Here, a 1 denotes that x is located in the array A at this position, and -1 that it is not. Now the number of intervals in which x is dominant is equal to the number of intervals in B which have a positive sum. This can be counted by making an array of prefix sums (call it C) and for each element of C we count how many elements in C before it are smaller than it. However, we do not need to iterate over all elements in C. For a block of -1's in B we can calculate at once the number of intervals with positive sum for which the right end is in this block. This can be done with a segment tree with propagation, where the nodes store, apart from the usual sum, the sum of the elements multiplied by $k, k-1, \ldots, 2, 1$ $(k \cdot first\_element + (k-1) \cdot second\_element + ... + 1 \cdot last\_element)$, where k is the number of elements covered by the segment tree node. For more details see the official implementation. The complexity of this solution is $O(n \log n)$, but to score all of the points it was possible to have other complexities, such as $O(n\sqrt{n})$.

Task Parkovi

Prepared by: Dominik Fistrić
Necessary skills: binary search, greedy algorithms

The first subtask can be solved by iterating over all choices of k parks and calculating the distances to the remaining nodes.

In the second subtask, the desired node is precisely the tree center, which can be found as one of the two central nodes on the tree diameter. The diameter of the tree can be found in a standard way, by calling a DFS two times to find the furthest node to the furthest node to some starting node.

Solving the third and fourth subtasks required binary search. In the third subtask, one iteration of binary search is just a simpler case of the general case of a tree. Therefore, we'll just describe the general case.

We do a binary search on the answer. Fix some r and now we need to determine if it's possible to place k parks so that each node has at least one park at a distance $\leq k$. We'll solve a somewhat stronger problem - we'll calculate the smallest number of parks needed to cover the entire tree, and compare this number to k. Intuitively, it is not beneficial to take the leaves, but to try to take nodes as far from them as possible, but to still cover them.

The algorithm is the following: we maintain a tree of still unprocessed nodes. In each step we choose some leaf. Call it u and let's say it's connect to a still unprocessed node v. We'll update some information for v and remove u. Additionally, it's possible that one of u or v is declared a park. This process is repeated until no more nodes are left.

We will refer to the nodes that were already processed and removed from the tree as being *outside*. The unprocessed nodes are divided up into two categories:

1. Nodes which have already been covered by someone from the outside. For such nodes we maintain an array $\texttt{dist}[x]$ representing the distance to the nearest park from the outside. This means that toward the inside we can still cover nodes up to a distance of $r - \texttt{dist}[x]$.
2. Nodes which have not yet been covered from the outside and for which we decided that they will be covered at some point in the future. For such nodes we maintain $\texttt{reach}[x]$ representing how far inward can we place a park so that it still covers x along with all the nodes we haven't yet covered and which depend on x to be covered.

At each step we remove a leaf u and for it's neighbour v we update $\texttt{dist}[v]$ or $\texttt{reach}[v]$. When doing this, it's possible that the category in which v belongs to changes. How to exactly update this information for v depends on a few simple cases and is left as an exercise to the reader. Let's illustrate one such case. If u and v are both already covered from the outside, and the length of the edge between them is w, then we set $\texttt{dist}[v]$ to $\min(\texttt{dist}[v], \texttt{dist}[u] + w)$. What remains is to determine when a node should be declared a park. We do this when a leaf u is not covered from the outside and when $\texttt{reach}[u] \le w$. If there is a strict inequality, we place a park in u, and otherwise we can place it in v.

The correctness of this algorithm follows from the fact that the parks are being placed in the last possible moment, i.e. when we don't have a choice. The time complexity of this procedure is $O(n)$ making the total complexity $O(n \log(n \cdot w_{max}))$.

Task Šarenlist

Prepared by: Pavel Kliska
Necessary skills: tree traversal, inclusion-exclusion principle, bitmasks

Let's look at the set S and a familiy of sets $A_i, i \in \{1, \dots, m\}$:

$$S = \{\text{set of all tree colorings}\}$$
$$A_i = \{\text{set of all colorings in which the path between } c_i \text{ and } d_i \text{ is monochrome}\}$$

In terms of these sets, the solution is the size of the set difference between S and the union of all the A_i's. From the inclusion-exclusion formula, we have:

$$\left| S \setminus \bigcup_{i=1}^{m} A_i \right| = \sum_{I \subseteq \{1,\dots,m\}} (-1)^{|I|} \left| \bigcap_{i \in I} A_i \right|$$

We use the notation $\bigcap_{i\in\emptyset} A_i = S$.

Since $m \le 15$, it's possible to iterate over all the 2^m subsets $I \subseteq \{1, \dots, m\}$. For each such subset we have to determine the size of $\bigcap_{i \in I} A_i$, i.e. the number of colorings in which each of the paths from c_i to d_i is monochrome, for each $i \in I$.

The number of such coloring depends on s, the number of components of the subgraph containg the monochrome paths, and the number of edges c which do not belong to any monochrome path. The desired value is then k^{s+c}. To see this, note that each component (i.e. a set of paths connected by shared edges) can be colored independently into one of k colors. Each edge not on one of the paths can also be colored in any of the k colors.

The value of s can be determined by constructing a graph in which the node i represents the path between nodes c_i and d_i, and an edge between i and j means that the paths between c_i and d_i and between c_j and d_j have some edge in common. By using a DSU structure or a DFS we can determine s in time complexity $O(m^2)$ for each I.

To calculate the value of c, we can determine how many edges are in the union of the paths from c_i to d_i for $i \in I$, and subtract this from the total number of edges, $n - 1$. For each path from c_i to d_i we maintain the set of edges it is made up from. Since there are at most $n - 1 \le 64$ edges, we can use a bitmask to store these sets, and then a union corresponds to the *bitwise or* operator $|$. Then, with $O(mn)$ preprocessing we can determine the value of c in $O(m)$.

Adding the complexities of c and s, and multiplying this by the total number of subsets of I, yields a final time complexity of $O(2^m m^2)$. Note that if $n - 1 > 64$, calculating c now takes $O(mn \log n)$ time which is too slow. The problem can still be solved for $n \le 10^5$, but this is left as an exercise to the reader.