Editorial

Tasks, test data and solutions were prepared by: Adrian Beker, Marin Kišić, Daniel Paleka, Ivan Paljak, Tonko Sabolčec, Bojan Štetić i Paula Vidas. Implementation examples are given in attached source code files.

Task: Paint

Prepared by: Tonko Sabolčec, Marin Kišić
Necessary skills: graph theory, *union find* data structure, *smaller-to-larger* technique

The first subtask can simply be solved by traversing the monochrome neighbourhoods using a DFS or BFS. In that case, the complexity of one coloring operation is linear with respect to the number of pixels in a component (monochrome neighborhood). In the worst case, the component will have $O(R \cdot S)$ pixels, so the total time complexity of this approach equals $O(R \cdot S \cdot Q)$.

The second subtask is a special case in which the image is one dimensional. Some of the approaches are:

- Maintaining a set of components (continuous intervals) using an efficient data structure (such as `std::set` in C++) that enables fast retrieval of neighbouring elements and their merging (insertion and deletion).
- Maintaining a set of connected components using *union find*.
- Dividing an array into parts of length $O(\sqrt{S})$ and keeping a flag for each part which tells us whether the whole part is a single component (i.e. is it monochrome).

The third subtask is special in the following way: in each step either there will be no changes (bucket tool is filled with the same color as the pixel it is applied on) or the component of the pixel will change its color and connect itself with *all* of its neighboring components. Maintaining these components can be done using the *union find* data structure, along which we will keep track of a set of active neighbors (e.g. using `std::set` in C++). When traversing the adjacency list of the current component, i.e. when connecting it with one of its neighbours, we delete that neighbor from its set of active neighbors (and vice versa). There will be no more than $O(R \cdot S)$ connections and each has a complexity of deleting an element from the set, $O(\log(R \cdot S))$. The total time complexity of this approach is therefore $O(R \cdot S \cdot \log(R \cdot S))$.

On first glance, it may seem that the algorithm from the third subtask can be used to obtain the full score. But, the problem resides in the fact that we don't necessarily need to connect the current component to all of it's neighbors in a single application of the bucket tool. Perhaps we can solve the problem by keeping the neighbours somehow grouped by color (e.g. using `std::map` in C++), but when changing color of a current component, we would still need to traverse over all of its neighbors in order to let them know we have changed our color. Let's keep all of these observations in mind, but return to the solution of the first subtask.

Note that, if the average size of components was sufficiently small, say 10 pixels, then the algorithm from the first subtask would be fast enough, i.e. $O(R \cdot S + Q \cdot 10)$. What if the average size of components was larger? Something like 100 pixels should be good enough as well, a 1000 should also be squeezable. What about 10000 pixels? Unlikely. Luckily, increasing the size of an average component decreases the number of components that have at least that many pixels. In other words, at each moment there are at most $R \cdot S/K$ components that have K or more pixels (we will choose the exact value of K later). This property could be of use. The intuition is as follows: we will use one of the naive approaches to solve for components with K or less pixels, and some other approach to solve for components with more than K pixels. The important thing is that this approach doesn't need to be efficient when the component has less than K pixels.

More precisely, for each component we track the following values: *color*, *size* and *a set of large neighbors* (those with more than K pixels – note that there is at most $R \cdot S/K$ of those). When connecting two components, we use *union find* to maintain these values. Additionally, for large components we keep around a *list of its neighbours grouped by color*. We will explain the algorithm using the following three operations:

1. Operation $\mathit{Color}(x, y, \mathit{color})$ – applies the bucket tool on a pixel (x, y):
   - Component label c to which pixel (x, y) belongs to is determined.
   - We use the operation $\mathit{NeighborsInColor}(c, \mathit{color})$ which determines the labels of neighboring components of the given color, $c'_1, c'_2, ..., c'_l$.
   - These components (c'_i) are connected with the current component c $(c = \mathit{Spoji}(c, c'_i))$.
   - After connecting, we go through all big neighbors of a component and we notify them on the color change (i.e. the label of a current component c is added into lists of neighboring components for a given color).
2. Operation $\mathit{NeighborsInColor}(c, \mathit{color})$ – finds neighboring components of a given color:
   - If component c is small, traverse all of its neighbors (at most $O(K)$ of them) and for each we check what color is it.
   - If component c is big, we use a list of neighbors grouped by color.
     Observation: After traversing that list, it can (has to) be deleted (after connecting, there will no longer be a neighbor of a given color – they will be a part of the same component).
3. Operation $\mathit{Connect}(c_1, c_2)$ - connects components c_1 and c_2:
   - Size of the resulting components equals to the sum of sizes of components c_1 and c_2.
   - The list of big components is obtained using *small-to-large* technique. (It's possible that after this operation we have some labels in our list that have been destroyed in the meantime, i.e. became connected in some larger component. We leave this implementation detail to the reader to resolve). the implementation detail discussed further in the text).
   - In case of connecting two large components, we need to obtain a joint list of neighbors grouped by color. This is also done using *small-to-large* technique.

Operation *Connect* will be done at most $O(R \cdot S)$ times. Smaller to larger technique secures the total amortized complexity of $O(R \cdot S \cdot \log(R \cdot S))$. The complexity of all operations *NeighborsInColor* in case of small components is $O(R \cdot S \cdot K)$, while the amortized complexity in case of big components is $O(R \cdot S)$. Finally, the complexity of all *Color* operations is $O(Q \cdot R \cdot S/K)$. The total complexity also depends on a parameter K and equals $O(R \cdot S \cdot \log(R \cdot S) + R \cdot S \cdot K + \frac{Q \cdot R \cdot S}{K})$. It follows that the optimal runtime should be obtained when $K = O(\sqrt{Q})$.

Task: Pastiri

Prepared by: Adrian Beker, Paula Vidas, Daniel Paleka
Necessary skills: graphs, breadth/depth-first search (BFS/DFS)

Denote the nodes of the three by V. We identify a sheep/shepherd with the node it occupies.

Notice that the task can be seen as an instance of the so called *set cover* problem: For every node $v \in V$, consider the set S_v of sheep which would be protected if we put a shepher in node v. Then we need to find the smallest subset $P \subseteq V$ of nodes such that $\bigcup_{v \in P} S_v$ equals the set of all sheep. The general set cover problem is NP-complete, but the specific tree structure here will help us to solve the problem in polynomial time.

Let's start with the first subtask, the case when the tree is a chain. A shepherd can protect only the first sheep to the left and/or to the right, so the family $\{S_v \mid v \in V\}$ contains the following subsets:

- $\{x\}$ for every sheep x;
- $\{x, y\}$ for consecutive sheep x, y such that $x - y$ is even.

We proceed with the following greedy algorithm: look at the first sheep. if the second one has the same parity, place a shepherd on the node halfway between them, otherwise place a shepherd in the same node as the first sheep. Next, remove the protected sheep and repeat the same algorithm. This solution has complexity $\mathcal{O}(N + K)$.

To solve the second subtask, we make use of representing the subsets of sheep by bitmasks in $\{0, 1, \ldots, 2^K - 1\}$, We first find the sets S_v for every node $v \in V$ in $\mathcal{O}(K \cdot N)$ by running breadth-first search from every sheep.

We can then solve this set cover instance by dynamic programming. For each bitmask *mask*, let $f(mask)$ denote the minimum number of sets S_v which cover *mask*. We iterate through the states *mask* in increasing order. When calculating $f(mask)$, we iterate over all $submask \subseteq mask$, and eash time *submask* corresponds to some of the sets S_v, we recalculate $f(mask) := f(mask \wedge submask) + 1$. To finish the transition, we put $f(submask) := f(mask)$ for all $submask \subseteq mask$,

Memory complexity is $\mathcal{O}(N + 2^K)$, and time complexity is $\mathcal{O}(K \cdot N + 3^K)$. (We leave the proof as exercise.)

For the remaining subtasks, we first pick an arbitrary node as the root. For each sheep x, we refer to the set $\{v \in V \mid x \in S_v\}$ as its *territory*. We say that two sheep x and y are *friends* if their territories have nonempty intersection. The main idea is to greedily place a shepherd that protects some sheep and all of the (currently) unprotected friends of that sheep. The following claim will help us with that:

Claim 1. For some sheep x, let $a(x)$ be its highest ancestor that is in its territory. Then, $S_{a(x)}$ contains all friends of x that are not deeper that x. *Proof.* Let y be a friend of x that is not deeper than x. If y is in the subtree of $a(x)$, the claim is obvious, so assume the contrary. Take some node z that is not part of territories of x and y, and let w be the midpoint of the path between x and y. We have

$$d(z, x) = d(z, y) = d(z, w) + d(w, x) = d(z, w) + d(w, y),$$

where $d(u, v)$ denotes the distance between nodes u and v. Also, for every sheep t it's true that

$$d(z, w) + d(w, x) = d(z, x) \leq d(z, t) \leq d(z, w) + d(w, t),$$

which implies $d(w, x) \leq d(w, t)$, and $d(w, y) \leq d(w, t)$ is proved analogously. Hence, w is contained in the intersection of territories of x and y. Moreover, since y is not deeper than x, w is an ancestor of x, so it must be located on the path from x to $a(x)$. Since y is not contained in the subtree of $a(x)$, we have

$$d(a(x), y) \leq d(w, y) = d(w, x) \leq d(a(x), x),$$

so a shepherd in $a(x)$ protects y, as needed. $\square$

According to Claim 1, the following algorithm is correct:

- Repeat until all sheep are protected:
  - Place a shepherd in $a(x)$, where x is (one of) the deepest currently unprotected sheep.

The straighforward implementation in the complexity $\mathcal{O}(N(N+K))$, is sufficient to solve the third subtask. To solve the fourth subtask, we need to speed up the algorithm. For each node v, let $dep(v)$ and $dist(v)$ denote the depth of the node and the distance to the closest sheep, respectively. We can calculate $dist$ by running a BFS starting from each sheep. Alternatively, we can imagine that we added a *dummy* node connected to all the sheep and run the BFS starting in that node. The following fact will help us to efficiently determine $a(x)$ for each sheep x:

Observation 2. If v is an ancestor of x, then $dist(v) \leq dep(x) - dep(v)$, where equality is attained if and only if v is in the territory of x.

Due to the above, $a(x)$ is the index of first maximum of $dist(v) + dep(v)$ over all v on the path from the root to x. Thus we can calculate $a(x)$ for each sheep x by a simple depth-first search from the root.

The only thing left is maintaining the deepest unprotected sheep efficiently.

If we sort the sheep by decreasing depth, we can reduce this problem to maintaining the set of protected sheep.

Consider the directed graph G with the same vertex set V as the given tree, and with edges (u, v), where $\{u, v\}$ is an edge from the given tree, directed such that $dist(v) = dist(u) + 1$.

Observation 3. For each node $v \in V$, S_v is exactly the set of sheep x such that there exists a path in G from x to v.

Whenever we place a new shepherd, we can run a DFS from that node following edges of G backwards, ignoring the nodes that were already visited. According to the above, the sheep is protected if and only if it was visited by the DFS. Since each node is visited at most once, the complexity of this part is linear.

The time complexity of the solution is $\mathcal{O}(N + K \log K)$. Notice that the factor $\log K$ comes from sorting the sheep by depth, which is of course possible to do in complexity $\mathcal{O}(N + K)$ because the depths are at most N.

Therefore, it's possible to solve the task in linear complexity. For implementation details, see the official source codes.

Task: Semafor

Prepared by: Bojan Štetić, Ivan Paljak, Daniel Paleka
Necessary skills: matrix multiplication, graph theory, binary exponentiation

We will solve the harder case M=2.

We represent a state of the board by a bitmask of 10 bytes. Call a bitmask *nice* if the represented board shows a valid number.

It is well known that the vertices of the *hypercube graph* are all bitmasks of some size (10 here), and the edges connect bitmasks which differ in exactly one place. Let H denote the adjacency matrix of the hypercube.

Then the problem, formally, asks for the following: for each nice bitmask Y, find the number of walks of length N from X to Y of length N in the hypercube, such that the walk visits the set of nice bitmasks every K steps.

Claim 1: The number of walks of length D in the hypercube, starting in X and ending in Y, equals the (X, Y)-th entry in the matrix H^D.

Proof: Look into Daniel A. Spielman's "Spectral and Algebraic Graph Theory", brilliant stuff. For an elementary proof, it's Lemma 3 on the link:
https://courses.grainger.illinois.edu/cs598cci/sp2020/LectureNotes/lecture1.pdf $\square$

Definition 2: Let B be the principal submatrix of H^K indexed by only the rows and columns of the nice bitmasks.

Claim 3: The number of walks (from X to Y) of length N in the hypercube, where K divides N, such that every K steps the walk visits a nice bitmask, equals the (X, Y)-th entry in the matrix $B^{N/K}$.

Proof: Left as exercise. $\square$

There are several steps in the solution. We need to calculate the matrix B, then calculate $B^{N/K}$, and then multiply it with the matrix corresponding to the last $N \mod K$ steps. We omit the third because it is very similar to the first step, so we assume K divides N in the rest of the editorial.

Let us solve the second step (raising B to the N/K-th power) first. It is enough to use naive binary exponentiation and multiply two matrices in the ordinary cubic complexity, because for $M = 2$ we are dealing with 100×100 matrices.

The first step is tougher, because B is a submatrix of H^K, and H is a 1024×1024 matrix for $M = 2$. Naive calculation of H^K passes only the subtask $M = 1$. For smaller K and $M = 2$, it can be done with clever dynamic programming, and this should give the first four subtasks.

For the full problem, we need to raise the hypercube matrix to some power faster.

Claim 4: The (X, Y)-th entry of the matrix B depends only on the number of bits in which the bitmasks X and Y differ.

Proof: Without loss of generality, we can XOR all vertices of the hypercube with some fixed bitmask. Thus, we can assume $X = 0$. It's also clear that the order of the bits doesn't matter at all. Hence, the number of walks from 0 to Y depends only on the bitcount of Y.

Claim 4 shows that's it's enough to calculate the first row of the matrix H^K. Depending on the exact implementation, it can pass the first five subtasks, or even pass the whole problem. (Ask Dorijan Lendvaj for the XOR-convolution solution of quadratic complexity that runs faster than the official solution.)

The official solution calculates the numbers of walks starting in $X = 0$ in a smarter way. It's enough to calculate, for every $0 \leq r \leq 10$, the number of walks starting from bitcount 0 and ending in bitcount r.

The number of ways in which we can get from one bitcount $0 \le r \le 10$ to another is given by the following 11×11 matrix:

$$C = \begin{bmatrix} 0 & 10 & 0 & 0 & \dots & 0 & 0 & 0 \\ 1 & 0 & 9 & 0 & \dots & 0 & 0 & 0 \\ 0 & 2 & 0 & 8 & \dots & 0 & 0 & 0 \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ 0 & 0 & 0 & 0 & \dots & 9 & 0 & 1 \\ 0 & 0 & 0 & 0 & \dots & 0 & 10 & 0 \end{bmatrix}$$

By raising the matrix C to a suitable power, we can get the number of walks of length K from bitcount 0 to each bitcount $0 \le r \le 10$. To get the actual number of walks in the hypercube, we just need to divide by a suitable binomial coefficient, due to symmetry. For implementation details, see the official source code.

There is an even faster solution, which uses *exponential generating functions*. The number of walks of length K from the bitmask 0 to some bitmask with $10 - r$ bits equals the coefficient next to $\frac{x^K}{K!}$ of the following expression:

$$\left(\sum_{i=0}^{\infty} \frac{x^{2i}}{(2i)!}\right)^r \left(\sum_{i=0}^{\infty} \frac{x^{2i+1}}{(2i+1)!}\right)^{10-r} = \left(\frac{e^x + e^{-x}}{2}\right)^r \left(\frac{e^x - e^{-x}}{2}\right)^{10-r}.$$

By expanding the expression carefully, we get a solution of the complexity $O((5 \cdot M)^2 + (5 \cdot M) \log K)$, that is, calculating entries of the $n \times n$ hypercube matrix raised to some power K is possible in time $O(\log n \log K)$.

For similar ideas, look into Herbert Wilf's free book *generatingfunctionology*.

Task: Zagrade

Prepared by: Paula Vidas
Necessary skills: stack, parentheses, ad hoc

We first describe the solution for the case when the whole password is a valid sequence. In the first subtask we have $Q = \frac{N^2}{4}$, so we can query every even length interval (note that it makes no sense to query odd length intervals). One possible solution is the following: we ask for the first two charactes, first four, and so on until we get a positive answer. We then know the position of the close parenthesis which is paired up with the first open parenthesis. Now we can recursively solve the same task for the interval between these parentheses, and for the interval that is on the right of the close parenthesis.

In the third subtask, we can ask $Q = N - 1$ queries. We will use a stack, which is empty at the beginning. We traverse the positions in order. If the stack is empty, push the current position on the stack. Otherwise, ask the query for the interval between the position on the top of the stack and the current position. If the answer is positive, those parentheses are paired up (left is open and right is close), and we pop the stack. If the answer is negative, we push the current position on the stack. In the end, the stack will be empty.

What if the whole password is not a valid sequence? The algorithm is the same, but the stack doesn't have to be empty in the end. First half of the remaining postions must have a close parenthesis, and the second half an open parenthesis. We leave the proof of this fact as exercise.