Day 1: a contest from U of Wrocław

SOLUTION SKETCHES

A Azrael

We are given a directed graph, in which every edge has its (integer) capacity. Additionally, every edge outgoing from the source s has its corresponding coefficient $c(s, v)$. We are interested in a maximum flow from s to t with the smallest possible cost. The cost is defined slightly differently than usually: we are only paying for the flow on the edges outgoing from s, and if there are f units of flow through such an edge (f does not need to be integer), we are adding $c(s, v)f^2$ to the total cost.

Finding a maximal flow is easy: it is enough to use, say, the Edmonds-Karp algorithm. However, adding the restriction that its cost (defined as above) should be the smallest drastically changes the situation, for example the optimal flow no longer needs to be integer.

Let us first assume that $c(s, v) = 1$ for every v. The following construction will be useful in the following reasoning. Every edge $s \rightarrow v$ with capacity c is replaced with multiple edges with smaller capacities. In more detail, we choose a parameter t and create ct new edges. The capacity of the i-th edge is set to $\frac{1}{t}$, and its cost to $[g(\frac{i}{t}) - g(\frac{i-1}{t})]$, where $g(x) = x^2$. We obtain a new graph, in which the cost of every flow is defined in the "standard" way. We notice that the optimal solution has the property that we will never send a nonzero flow through the i-th edge corresponding to original $c(s, v)$ (in short: the i-th edge), but we are not sending any flow through the $(i - 1)$-th edge. Consequently, costs of the optimal flows in the original and the new graph differ by at most $\frac{X}{t}$, where X depends on the original graph, and therefore by increasing t we can obtain better and better approximations of the answer, and the optimal answer in the limit.

Let us consider an execution of the algorithm that find the cheapest maximum flow in the new graph by repeatedly finding a cheapest augmenting path. Because the cost are only defined on the edges outgoing from the source, the algorithm can be thought of as considering these edges in the order of non-decreasing costs, and sending as much as possible through the current edge. Let us modify the algorithm so that, if we cannot fully use the capacity of the current edge, we are not sending anything at all through it (note that this does not break the property that, if we are sending something through the $(i + 1)$-th edge then we are also sending something through the i-th edge). This change might increase the cost of an optimal flow, but as before taking sufficiently large t brings us as close to the optimal solution as we wish (this is slightly inaccurate: the change also decreases the maximum flow, but still in the limit we obtain the right answer).

Now we are ready to construct the final algorithm. For every $i = 1, 2, \ldots$ we find the maximum number ℓ of i-th edges whose capacities can be fully used. We observe that this number if nonincreasing for increasing values of i. Thus, instead of fixing t we can simulate the behaviour of the final algorithm for a very large (think infinite) value of t as follows. For $\ell = n, n - 1, \ldots$ find the largest $\epsilon > 0$ such that we can simultaneously send additional ϵ units of flow through each of ℓ edges outgoing from the source. In other words, repeat the following: find the largest $\epsilon > 0$ such that we can send can simultaneously send additional ϵ units of flow through all edges outgoing from the source from which it is still possible to reach the sink (in the residual network).

Finally, let us consider the general case in which $c(s, v)$ might be different for different v. Then, the i-th edge might have different costs for every v, so we can't consider all of them together, and this was the gist of the above argument. However, we can modify the definition of the i-th edge as follows. The capacity of the i-th edge corresponding to $s \rightarrow v$ with capacity c is $\frac{1}{c(s,v)t}$, and its cost is $c(s, v)[g(\frac{i}{t}) - g(\frac{i-1}{t})]$. Now the cost of fully using all i-th edges is the same, but they have different capacities. To deal with this, we can modify our solution as follows: as long as possible, find the largest $\epsilon > 0$ such that we can simultaneously send $\frac{\epsilon}{c(s,v)}$ additional units of flows thought every edge $s \rightarrow v$ from which it is still possible to reach the sink (in the residual network).

B Brainy

For a given prime number k we want to construct an array of size $k^2 \times k^2$ in which every entry contains a number from $\{1, \ldots, k\}$. We require that every possible array of size 2×2 in which every entry contains a number from $\{1, \ldots, k\}$ occurs exactly once in our array. Here we think that our array "wraps around" both horizontally and vertically.

Let us begin with constructing a cyclic de Bruijn of length k^2 over the alphabet $\{1, \ldots, k\}$, call it S. All substrings of length 2 are distinct in S. Let us denote by $S^{(d)}$ the cyclic shift of S by d positions. As our solution we print an array in which every row is an appropriately chosen cyclic shift of S, let us denote them as $S^{(d_0)}, S^{(d_1)}, \ldots, S^{(d_{k^2-1})}$, where $d_0 = 0$. We want to ensure that all differences $(d_{(i+1) \bmod k^2} - d_i) \pmod{k^2}$ are unique (for $i = 0, 1, \ldots, k^2$). This will guarantee that the required property concerning all possible arrays of size 2×2 indeed holds. To this end, we define $d_{i+1} = d_i + i \pmod{k^2}$ for $i = 0, 1, \ldots, k^2 - 2$. For this construction to be correct we need $\sum_{i=0}^{k^2-1} i = 0 \pmod{k^2}$, which indeed holds for every odd k. The case $k = 2$ can be solved with pen and paper.

C Calendar

We need to perform a cyclic shift of an array $A[0 \dots n-1]$ by k positions. In every step we can reverse a segment of the array and need to minimise the number of steps. Let B^R be the array B reversed and with capital letters we denote arrays and small letters are single elements. It is not hard to see that we can always perform the cyclic shift in at most 3 steps but there are some corner cases to consider:

- If $n = 1$ or $k = 0$ we don't have to do anything.
- If $n = 2$ and $k = 1$ we simply reverse the array.
- For $k = 1$ we sort in two steps: $A = Bb \to bB^R \to bB$. Similarly for $k = n - 1$.
- (**most tricky**) For $k = 2$ we can sort in 2 steps: $A = Bcd \to cB^Rd \to cdB$. Similarly for $k = n - 2$.
- Otherwise let $A = BC$ where C has k elements. Then we sort in 3 steps: $A = BC \to C^RB^R \to CB^R \to CB$.

D Denominations

We want to compute $D_{\{1,5,10,25\}}[n]$ where $D_S[n]$ is the number of ways we can give change of n with unlimited number of coins from set S. We start with $D_{\{1\}} = 1$ and $D_{\{1,5\}} = \lfloor \frac{n}{5} \rfloor + 1$. For bigger sets we can use the classic dynamic programming:

$$D_{S\cup\{a\}}[n] = \sum_{i=0}^{\lfloor \frac{n}{a} \rfloor} D_S[n - a \cdot i], \quad \text{for } a \notin S$$

Having the formula for $D_{\{1,5\}}$, we use the above property with $a = 10$. By carefully rearranging the terms we obtain: $D_{\{1,5,10\}} = (\lfloor \frac{n}{10} \rfloor + 1)(\lfloor \frac{n+5}{10} \rfloor + 1)$. To compute $X[n] = D_{\{1,5,10,25\}}[n]$ we proceed similarly, but the calculations and the final formula are more involved, though still computable in $\mathcal{O}(1)$ time. As $X[n] = X[n - (n \bmod 5)]$, it is useful to consider the formula separately for $n \equiv 0 \pmod{10}$ and $n \equiv 5 \pmod{10}$.

There are solutions running in $\mathcal{O}(\log n)$ time, but they shouldn't fit in the time limits.

E Euclid

Given 3 points P_1, P_2, P_3 in 3D we need to find a point P that minimizes the sum of distances to the given 3 points. Let S be the plane passing through all points P_i. Observe that P should lie on S, otherwise it's projection on S would have smaller sum of distances. Hence we can reduce the problem only to S, in two dimensions.

In the plane, the point minimizing the sum of distances from all vertices of a given triangle T is known as Toricelli point. When T has an angle greater than $120°$, the sought point is in the obtuse angle. Otherwise we choose two arbitrary sides of T and put an equilateral triangle on each of them, outside T. For each of the two chosen sides e we draw a line from the new vertex of the equilateral triangle on e to the vertex of T opposite e. Then the Toricelli point lies on the intersection of the two lines.

Other solution makes use of the fact that the distance function: $f(x, y, z) = \sum_i |P_i - (x, y, z)|$ is convex for each of the coordinates, so we can run ternary search for each of the coordinates separately (or in other words: nested 3 times) and find the optimum.

F Flowers

The input to this problem is a tree on $n = 3$ nodes. We want to color the nodes with three colours (red, green, blue) so that the number of nodes of each colour is the same (exactly k), and the colouring is... a proper colouring: for every two nodes u and v that are neighbours the color of u is different than the colour of v. Even though the problem statement asks you to find such a colouring or output that there is none, it turns out that for trees in which the degree of every node is bounded by 4 a solution always exists!

Let us call the sought colouring an equi-colouring, and prove that it indeed always exists. The proof will be constructive, and results in a linear-time colouring algorithm.

First, consider a simpler case where the degree of every node is at most 3. We begin with colouring all nodes using only two colours (red and blue). Could it happen that we have very few nodes of some colour? Let A be the set of nodes coloured red, and B be the set of nodes coloured blue, and by symmetry assume $|A| \leqslant |B|$. We have the following lemma.

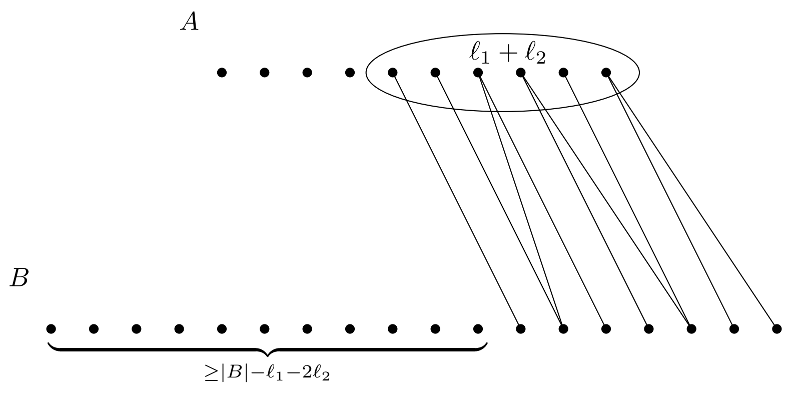

Figure 1: Red nodes of degree $\leqslant 2$ can't block too many blue nodes.

Lemma 1. *$\frac{1}{3}n \leqslant |A|$ and $|B| \leqslant \frac{2}{3}n$.*

Proof. It is enough to show that $\frac{1}{3}n \leqslant |A|$. There are exactly $n-1$ edges in our tree, each edge connects a red node and a blue node. As the degree of every node is at most 3, we must have at least $\frac{n-1}{3}$ red nodes. As this is an integer number and n is divisible by 3, the number of such nodes is in fact at least $\frac{1}{3}n$. □

To obtain a proper equi-colouring, we will recolour some of the nodes. More specifically, we will make some nodes green. For example, if $|A| = \frac{1}{3}n$ this is very easy: we just need to partition B into two subsets of equal size. On the other hand, if $|A| > \frac{1}{3}n$ then we need to make exactly $|A| - \frac{1}{3}n$ red nodes green. But which nodes should be chosen? Choosing u for recolouring means that we "block" all of its neighbours in B: we won't be able to recolour them. Intuitively, it seems best to recolour nodes with the smallest degrees. It turns out that it is always enough to consider nodes of degree 1 and 2 for recolouring.

Lemma 2. *Let ℓ_d b the number of nodes of degree ℓ_d in A. We have $\ell_1 + \ell_2 \geqslant |A| - \frac{1}{3}n$.*

Proof. We want to show that $\ell_{\geqslant 3} \leqslant \frac{1}{3}n$. There are $n-1$ edges in the whole tree, every node of degree $\geqslant 3$ blocks at least three of them, so we cannot have more than $\frac{n-1}{3}$ such nodes in A. Again, this really is $\frac{1}{3}n$. □

Consequently, recolouring $|A| - \frac{1}{3}n$ red nodes with the smallest degrees leaves us with at least

$$|B| - 2(|A| - \frac{1}{3}n) = n - 3|A| + \frac{2}{3}n = \frac{5}{3}n - 3|A|$$

non-blocked blue nodes. We would like to choose $\frac{1}{3}n - (|A| - \frac{1}{3}n)$ of them, but is this always possible?

$$\begin{aligned} \frac{5}{3}n - 3|A| &\geqslant \frac{2}{3}n - |A| \\ n &\geqslant 2|A| \end{aligned}$$

so, as $|A| \leqslant |B|$, we will always have enough non-blocked blue nodes!

Now we are ready to proceed with the general case, where the nodes could have degrees 4. Let us first try the above reasoning. If $|A| \geqslant \frac{1}{3}n$ then nothing bad happens. However it could happen that $|A|$ is smaller. What then? It can be observed that there must be many blue leaves.

Lemma 3. *If $|A| \leqslant \frac{1}{3}n$ then we have at least $|B| - \frac{1}{3}n$ blue leaves.*

Proof. Assume the opposite: there are more than $\frac{1}{3}n$ blue nodes of degree $\geqslant 2$. After removing all blue leaves we obtain a tree in which the number of red nodes is (strictly) smaller than the number of blue nodes, and further every blue node has at least two (of course red) neighbours. Let us try to prove that this is impossible. Take a smallest tree meeting both conditions. It surely contains some red leaf, let us consider its (unique) blue neighbour:

1. if it has degree two, we remove both nodes.
2. if it has degree strictly larger than two, we remove the leaf.

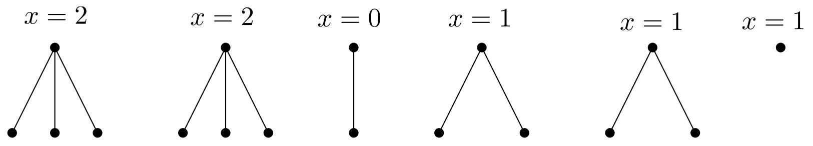

Figure 2: Forest in which all red nodes are leaves.

In either case, we obtain a smaller tree meeting both conditions, which is impossible by the assumption. □

Consequently, we can remove $\frac{1}{3}n$ red nodes from the tree to obtain a forest on $\frac{2}{3}n$ nodes, in which every red node is a leaf (additionally, the number of red nodes is at least as large as the number of green nodes). As the degree of every node is at most 4, the forest looks like shown in the figure. Every component can be coloured in two ways. Let us define the **balance** of a colouring as the different between the number of red nodes and the number of blue nodes. For every components, one colouring increases the balance by x and the decreases the balance by x, where $x \in \{0, 1, 2\}$ (depending on the shape of the component). We observe that, if there exists a component with $x = 1$, we can easily obtain a colouring with balance 0: just keep that component on the side, colour every other component by repeatedly choosing the colouring that minimises the absolute value of the balance, and finally use the component with $x = 1$ to make the balance zero.

The case when there is no component with $x = 1$ is left as an exercise.

G Garden

Given a sequence $a_1, a_2, \ldots, a_n$ we need to check if there exists (not necessary contiguous) super-increasing subsequence of length k, where sequence $b_1, b_2, \ldots, b_k$ is super-increasing iff $b_2 - b_1 < b_3 - b_2 < \ldots < b_k - b_{k-1}$ and $b_1 < b_2 < \ldots b_k$.

First we start with a slight simplification by observing that a sequence is super-increasing iff $0 < b_2 - b_1 < b_3 - b_2 < \ldots < b_k - b_{k-1}$. As for finding the longest **increasing** subsequence we use dynamic programming, maybe here we can use a similar idea? Let's say we found a prefix $a_{i_1} < a_{i_2} < \ldots a_{i_{k'}}$ of our desired super-increasing subsequence of length k. While choosing $a_{i_{k'+1}}$ we need to remember about two constraints:

1. $i_{k'} < i_{k'+1}$
2. $a_{i_{k'}} - a_{i_{k'-1}} < a_{i_{k'+1}} - a_{i_{k'}} \Leftrightarrow 2a_{i_{k'}} - a_{i_{k'-1}} < a_{i_{k'+1}}$

So there are only two important values for us: $i_{k'}$ and $2a_{i_{k'}} - a_{i_{k'-1}}$! Moreover, if we have two subsequenecs with the same value $i_{k'}$, we can disregard the one that has larger value of $2a_{i_{k'}} - a_{i_{k'-1}}$ (we call such expression *drop* of subsequence). This gives us a solid basis for construction of a solution based on dynamic programming that uses only $\mathcal{O}(nk)$ states. We are left to show how to quickly find the optimal solution for each of them. Of course we can only prolong each of the computed nk subsequences by every possible subsequent element, but we cannot do it: $\mathcal{O}(n^2k)$ is way too much.

Denote as $b[i][j]$ the smallest possible drop of super-increasing subsequence of subsequene of length j for which the last element is a_i. After a while of rewriting the terms we can deduce that for $j \geqslant 3$:

$$b[i][j] = \min\{2a_i - a_{i'} : i' < i \text{ and } b[i'][j-1] < a_i\} = 2a_i - \max\{a_{i'} : i' < i \text{ and } b[i'][j-1] < a_i\}$$

We need a smart way to find i' among $1, 2, 3, \ldots, i-1$, such that $b[i'][j-1] < a_i$ and $a_{i'}$ is as big as possible. We would like to repeat such an operation for all i, which suggests that we need to implement a data structure to which we can add pairs $(b[i'][j-1], a_{i'})$, and with which we can retrieve maximum second coordinate of points with first coordinate upper bounded. It turns out that to get $\mathcal{O}(\log n)$ running time for both the operations it's enough to use our favourite static binary tree. More precisely, first we normalize all first coordinates so that they are from $[1, n]$ and then we think about the first operation as replacement of the value of $A[i]$ and the second as the maximum on $A[1..j]$, which gives us the total running time $\mathcal{O}(nk \log n)$.

H Hamilton

Given a tree, we need to find a sequence of nodes of the tree such that every node appears exactly once and the distance between every two consecutive nodes is at most 3, we call such a sequence traversal. We will find even more

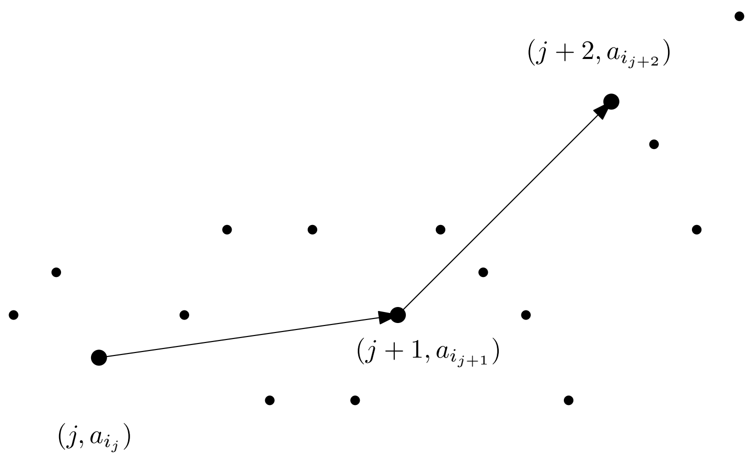

Figure 3: Left turn.

restricted traversal, namely require that we start in the root of the tree and end in one of its children (if exists). Clearly by reversing such a traversal we obtain a traversal that starts in a child of the root and ends in the root.

We construct the restricted traversal recursively. Let r be the root of the considered tree and its children are $c_1, \dots, c_k$. Let the reversed restricted traversals for subtrees of every c_i start with w_i and end in c_i. Root r and the concatenation of traversals $w_i \to c_i$ gives us a restricted traversal as the distance from r to w_1 is 2, distance from c_i to w_{i+1} is 3 and we finish in c_k that is a child of r.

I Invigilation

We are given a polygonal chain with a subset of its vertices distinguished and need to find the smallest number of cameras to put on the line $y = H$ (highway) that can observe all the distinguished points.

Every distinguished point can be visible from a segment of the highway. Having all the segments computed, we need to hit them with the smallest number of points, which can be done greedily by hitting the segments from left to right, as late as possible.

Now we show how to compute left endpoints of each segment. We proceed left-to-right and maintain the convex hull of the points visited. For every considered point p, intersection of the line passing through the last side of the convex hull and the highway gives us the leftmost point on the highway, from which p is visible. The convex hull can be maintained on the stack and constructed in overall $\mathcal{O}(n)$ time. We proceed similarly for the right endpoints, sort the intervals for distinguished points and hit them greedily, so the algorithm runs in $\mathcal{O}(n \log n)$ time.

J Joke

We are given a text T and a set of k forbidden patterns and need to erase some letters from the text so that there is no occurrence of a forbidden pattern. We replace a single occurrence of character α with a space at the cost c_α and need to minimize the total cost.

First we detect all occurrences of patterns in $\mathcal{O}(nk)$ time with KMP algorithm. This gives us a set of intervals such that from each of them we need to remove at least one position. We remove all intervals that have a smaller interval fully contained in them, which corresponds to removing forbidden patterns that include a smaller forbidden pattern. Then the obtained set of intervals $\mathcal{I}$ has the property, that after sorting them by left endpoints, right endpoints also get sorted.

Now we run dynamic programming for the sorted list of intervals: $dp[j]$ is the minimal cost of hitting all intervals $\mathcal{I}_1, \dots, \mathcal{I}_j$. Clearly $dp[j] \leqslant dp[j+1]$. We iterate over all positions i of letters to be erased. Erasing a letter at position i costs $c_{T[i]}$ and suppose it hits all intervals $\mathcal{I}_a, \dots, \mathcal{I}_b$. Then we can set $dp[x] := \min\{dp[x], dp[a-1] + c_{T[i]}\}$ for all $x \in [a, b]$. We can retrieve and update all values of dp in $\mathcal{O}(\log n)$ time using a segment tree with operations: `SetMin`(range,value) and `GetValue`(position). This gives us overall running time $\mathcal{O}(nk + n \log n)$.

K Klothes

We are asked to represent a given s as a sum of k **pairwise distinct** integer numbers from $[1, n]$. Let us start with two simple observations:

1. the smallest such number is $1 + 2 + \ldots + k = \frac{k(k+1)}{2}$,
2. the largest such number is $n - k + 1 + n - k + 2 + \ldots + n = \frac{k(2n-k+1)}{2}$.

Now it turns out that all numbers from $\frac{k(k+1)}{2}$ to $\frac{k(2n-k+1)}{2}$ can be represented as a sum of k pairwise distinct numbers from $[1, n]$. To prove this, assume that we already know how to find such a representation for some $s > \frac{k(k+1)}{2}$, then we can obtain a representation of $s - 1$ by decreasing one of the numbers. To find a representation of s, we can proceed as follows: if $s \leqslant \frac{2(n-1)-k+1}{2}$ then we decrease n by one, otherwise we include n in the representation, decrease s by n, and n by one.

L (Smurf)Land protection

We are given a directed graph and for every node we need to check if removal of the node strictly increases the total number of strongly connected components (SCC) in the graph. We can decompose the graph into SCCs and consider each of them separately, so now we describe how to process a single SCC S.

Consider an arbitrary node r in S. As S is strongly connected, every node in S is reachable from r and from every node in S we can reach r. Then removal of a node u increases the number of SCCs iff there is a node w such that w cannot be reached from r or we cannot reach r from w. As we can consider S and S with reversed edges, we reduced the problem to the following one: given a (root) node r, for every node u in the graph we need to check if after removal of u there exists a node that is not longer reachable from r. This is equivalent to checking for every node u if there is at least one child c of u such that all paths from r to c pass through u, we call such node critical.

We solve this problem with the following approach. Consider the tree T of DFS traversal from r and renumber nodes in pre-order. Let $F_z[u]$ to be the highest node on the path from r to u from which we can reach u using only nodes $z, z+1, \ldots, n$. Let $I(u)$ be the set of nodes from which there is an edge to u. For $z = n, n-1, \ldots, 2$ we compute F_z from F_{z+1} using the property:

$$F_z[z] = \min(\{u : u \in I(z) \wedge u < z\} \cup \{F_{z+1}[u] : u \in I(z) \wedge u > z\})$$

as either we reach z directly from an ancestor of z in T or using a path of nodes with pre-order greater than z. For all other nodes $w > z$, we have $F_z[w] \leqslant F_{z+1}[w]$. The only possibility that we need to update some $F_z[w]$ is when it is reachable by a path through z. This happens only for the nodes in the subtree of z in T and for each of them we need to set $F_z[w] = \min(F_{z+1}[w], F_z[z])$. As all nodes in the subtree of a node form an interval of pre-order indices, this update and subsequent lookups to $F_{z+1}[\cdot]$ can be performed in $\mathcal{O}(\log n)$ time using a segment tree. Finally, a node is critical iff $F_z[z]$ is the parent of z in T.

The above approach runs in $\mathcal{O}(m \log n)$ time. We note that using dominators we can obtain an $\mathcal{O}(m\alpha(m, n))$-time solution.