Problem A. Admissible Map

Problem author and developer: Ilya Zban

Let's call any string of shape "`RLRL...RL`" trivial.

Lemma: any non-trivial string has at most one constitution as an admissible map.

Proof: let's consider string $s_0 s_1 \dots s_{|s|-1}$, and i be a first even number such that $s_i s_{i+1} \neq$"`RL`". There can be a few cases:

- s_i="`L`". String s doesn't have a constitution as an admissible map, as either the i-th symbol will be on the left edge of the matrix (and its edge isn't directed to any other cell), or it points to the left, to "`RL`" so there are two incoming edges to neighboring "`L`" (or it points out of the matrix).
- s_i="`U`". String s doesn't have a constitution as an admissible map, as either the i-th symbol will be on the upper edge of the matrix (and its edge isn't directed to any other cell), or it points to some pair "`RL`", that should be matched to each other.
- s_i="`R`". As $s_{i+1} \neq$ "`L`", the only incoming edge to i-th cell can be "`U`". And we can notice that it can only be the first "`U`" in the string, as otherwise the first "`U`" would point either outside of the matrix or to some "`RL`" pair. So, if s_j is the leftmost "`U`", string s can only be constituted as $\frac{|s|}{j-i} \times j - i$ map.
- s_i="`D`". Let's say that $k = i$ if $s_{i+1} \neq$ "`L`", and otherwise take k as maximal number such that $s_{i+1} \dots s_k$ ="`LLL..L`". Substring $s_i \dots s_k$ should be in same row of matrix, and as $s_{k+1} \neq$"`L`", by same argument as above we can see that the first "`U`" in the string should point to s_k. So, string s can only be constituted as $\frac{|s|}{j-k} \times j - k$ map.

So, the shape $n \times m$ of a matrix for any non-trivial substring $s_l s_{l+1} \dots s_r$ is determined only by the position of the first "`U`" after l. We can compute an array m_l meaning that any substring $s_l s_{l+1} \dots s_r$ should be constituted as $\frac{r-l+1}{m_l} \times m_l$ matrix. This array can be computed in linear time directly from proof.

Using m_l we can iterate over all $r = l + t \cdot m_l$ for all t, and check all substrings $s_l s_{l+1} \dots s_r$. We need to be able to quickly determine if substring $s_i \dots s_{i+m_l-1}$ can be a top, middle or a bottom row of constituted matrix. It can be done in $\mathcal{O}(1)$ time.

Let's consider the case of the middle row (other cases are similar). We want to check that no edge from $s_i \dots s_{i+m_l-1}$ goes outside of the matrix and that each cell has exactly one incoming edge. First, we need to check that $s_i \neq$"`L`" and $s_{i+m_l-1} \neq$"`R`". Conditions on incoming edges can be tested using hashes. We choose a hash base x, and build 4 arrays $a^c_0 \dots a^c_{|s|-1}$ (c in "`ULDR`") such that $a^U_j = x^j$ if s_j ="`U`", $a^D_j = x^j$ if s_j ="`D`", $a^R_j = x^{j+1}$ if s_j ="`R`" and $a^L_j = x^{j-1}$ if s_j ="`L`", and all other values are zero. Then we can see that each cell from $s_i \dots s_{i+m_l-1}$ has one incoming edge if $x^{m_l} \sum\limits_{t=i-m}^{i-1} a^D_t + \sum\limits_{t=i}^{i+l-1} (a^L_t + a^R_t) + x^{-m_l} \sum\limits_{t=i+m}^{i+2m-1} a^U_t = \sum\limits_{t=i}^{i+l-1} x^t$. This check can be done in constant time using a precomputed array of prefix sums.

Using these checks we can iterate over all non-trivial substrings, test them and add missed trivial strings. This solution works in $\mathcal{O}(s^2)$.

We can further notice that for each l we iterate over all possible $\frac{|s|}{m_l}$ possible r-s with step m_l. We can count all l that have both the same m_l and $l \mod m_l$ together, as we just do a lot of duplicated work in that case. This optimization gives us a very fast solution that works in $\mathcal{O}(\sum\limits_{m=1}^{s} \frac{s}{m} \cdot \min(cnt_m, m)) = \mathcal{O}(s\sqrt{s})$ in worst case (here cnt_m is the number of different remainders $l \mod m_l$ for each m).

Problem B. Budget Distribution

Problem author and developer: Pavel Kunyavskiy

Let's first solve a single-topic problem. If there are no items that have too much assigned money (more they should have at the end, including unassigned money), the answer is 0. Otherwise, it's not important to which item money is assigned, if not giving too much money to any of the items. So, with a fixed set of items already having too much money, non-optimality function is linear-fractional in terms of extra money, i.e it has the form $\frac{ax+b}{cx+d}$. Also, it is easy to check that the derivative is increasing (it's negative and getting closer to zero), so overall the function is convex piecewise-linear-fractional function.

To solve the full problem, we need to find a way to combine solutions for different topics. Let f_i be non-optimality function for i-th topic. In fact, we need to find the function $f(x) = \min\limits_{\sum x_i = x, x_i \ge 0} f_i(x_i)$. As would be shown later, this function is convex piecewise-linear-fractional too, with a linear number of pieces.

To make a set of x_i locally-optimal, it should be impossible to find a pair of indices i and j, such that decreasing x_i a bit, and increasing x_j a bit would decrease the function. In fact, this means that all left derivatives should be less (greater by absolute value, as they are negative), than all right derivatives. Intuitively, if we think about the continuous process of distributing money, it should work like this: give next infinitely small amount of money to all topics with smallest derivatives (those, who are decreasing the fastest), in such proportion that their derivatives would be still the same. And the derivative of the resulting function in that point is equal to that minimum of the right derivatives.

So, let's construct a function by sweeping a line over its derivative. Each piece of the topics' function is working as "at this range of the derivatives, you also need to add this function". We know derivatives at each point, where switch from one piece to another happens, and the piece has a unique point with such derivative. So the only remaining part is to explicitly find the function for each part, when the set of topics we are giving money to is known.

For convenience, let $f_i(x) = \frac{a_i^2}{x-b_i} + c_i$. Any linear-fractional function with negative derivative can be written in such form. Intuitively, b_i and c_i are coordinates of the hyperbola center, and a_i is just decreasing speed factor. This form is convenient, as b_i and c_i can be just thrown away, as c_i only shifts resulting value by constant, and b_i only shifts argument by constant. So we can find the result for functions of the form $\frac{a_i^2}{x}$, and then just shift it, to make value and derivative in a point where it's connected to the previous part correct.

For these kinds of functions, making derivatives same, means that $\frac{x_i}{x_j} = \frac{a_i}{a_j}$ for all pairs of i and j. That means $x_i = \frac{a_i}{\sum a_i} x$. And the total function is $\frac{(\sum a_i)^2}{x}$.

Problem C. Connect the Points

Problem author and developer: Dmitry Yakutov

There are a lot of ways to solve the problem.

1. **Brute Force.** Take all X- and Y-coordinates of given points and build a "grid" of nine points. Take all 18 segments between them and iterate over all subsets of these segments. Choose the best one.

2. **Deal with "cases".** Mark three points with label sorted by X-coordinate. Then order points by Y-coordinate and watch the labels. There are six different orderings. It is possible to build the optimal set of segments for each ordering.

3. **Median point.** Let x_m be median X-coordinate among all the points and y_m be median Y-coordinate. $M = (x_m; y_m)$ is the median point. Note that M can either be in the input set or not. It is provable that you can connect all three points with M in the shortest way and output the resulting set of segments.

4. **Median segment.** Let x_m be median X-coordinate as before and y_{min} and y_{max} be minimum and maximum Y-coordinates. Take the segment from $(x_m; y_{min})$ to $(x_m; y_{max})$ and connect all three points with this segment by horizontal segments. It is also provable that such a set of segments is correct.

Problem D. Deletive Editing

Problem author: Dmitry Yakutov; problem developer: Roman Elizarov

This problem can be solved in a straightforward way. The key observation is that the order in which the letters are called out does not matter in this game. We only need to know how many times each letter is called out in order to go from the initial word s to the final word t.

So first, let us compute the number of occurrences of each letter from 'A' to 'Z' in both words s and t. Let's call them s_a and t_a for each letter a. Using these numbers, we can calculate how many times each letter shall be called in order to get a chance of getting to t. That is $s_a - t_a$ times for each letter a.

If $s_a - t_a < 0$ for any letter a, then the answer is "NO".

Otherwise, there is a chance for a positive answer. However, we also need to verify that the order of the letters in t is correct. The easy way to verify it is to simulate the game, dropping the first $s_a - t_a$ occurrences of each letter a, and then compare the result with t.

Problem E. Even Split

Problem author and developer: Egor Kulikov

First, let's find what is the minimal possible length of the longest segment. Using binary search on said length suppose we need to test if *len* is enough. We can show that it is necessary and sufficient if we can find n segments of length *len* (maybe intersecting) that cover the whole of Segmentland that can be bijected to n houses so that corresponding segment contains the corresponding house. This can be done greedily going from left to right and always trying to fit the next segment in the rightmost way so that we still cover some prefix of Segmentland and the corresponding house.

Now we also need to find the maximal possible length of the shortest segment. This is done similarly with binary search, only now we want to find segments of the same length that fit into Segmentland without intersection, and in our greedy algorithm we will fit them in leftmost way.

Given these two lengths, *min* and *max*, it can be shown that we can find a subdivision of Segmentland that adheres to these limits. One way to find such a subdivision is this:

- For $i \in 0 \ldots n$ let's have two parameters, $c_i \le d_i$, which means that the corresponding dividing point lies somewhere between c_i and d_i. We calculate those from left to right starting with $c_0 = d_0 = 0$, and $c_{i+1} = max(c_i + min, a_{i+1}), d_{i+1} = min(d_i + max, a_{i+2})$ (we assume $a_{n+1} = l$).
- For $i \in 0 \ldots n$ let's have ans_i as an actual boundary. We will calculate it from right to left, starting with $ans_n = l$, and on each step we will select any point from $[c_i, d_i]$ that is between *min* and *max* distance from ans_{i+1} (it is easy to see there will always be at least one such point).
- Now we have an answer — $s_i = ans_{i-1}, f_i = ans_i$.

Problem F. Fancy Stack

Problem author: Dmitry Gozman; problem developer: Gennady Korotkevich

Let's call the blocks on the even positions *big* $(b_2, b_4, \ldots, b_n)$, and the blocks on the odd positions *small* $(b_1, b_3, \ldots, b_{n-1})$.

First, assume that the block sizes are distinct. We'll process the blocks in decreasing order of size (i.e., let $a_1 \geq a_2 \geq \ldots \geq a_n$). To count the stacks, we will use dynamic programming.

Let $f_{i,j}$ be the number of ways to put i biggest blocks into their places so that j of these blocks are big, and $i-j$ of these blocks are small. As a base case, $f_{0,0} = 1$. We'll implement it as a forward DP, and to make transitions for the $i+1$-th block we will decide whether it's big or small.

If the $i+1$-th block is big, its position in the stack is determined uniquely (specifically, it's b_{n-2j}). Hence, we make a transition to $f_{i+1,j+1}$.

If the $i+1$-th block is small, there are $\max(j-1,0) + [j = \frac{n}{2}]$ possible places for small blocks (between any two big blocks, and at the top of the stack if all big blocks have been placed), out of which $i-j$ are already occupied. Note that all these potential places will be available for all future (smaller) small blocks. Hence, it doesn't matter which particular place we occupy with the $i+1$-th block. We make a transition to $f_{i+1,j}$ with coefficient $(\max(j-1,0) + [j = \frac{n}{2}]) - (i-j)$.

Now, if we allow blocks to have equal sizes, we can just process the blocks in groups of the same size — again, in decreasing order of size. In each group, at most one block can be large (since all big blocks must have distinct sizes). This way, we still have just two transitions from any DP state, this time with both of them using binomial coefficients — coming from the fact that we can choose any valid unoccupied places for small blocks.

Alternatively, instead of splitting the blocks into groups explicitly, we can keep the solution from the "all distinct" case, and only allow the last block in each group to be big (that is, block i can only be big if $a_i > a_{i+1}$).

The time complexity of this solution is $O(n^2)$.

Problem G. Global Warming

Problem author and developer: Nikolay Budin

Let's do a sweep plane algorithm from top to bottom. We will maintain connected components for points that are higher than the plane and areas of surfaces for these components. When the sweep plane meets a new point, we iterate over its neighbors that are higher than the plane and unite their connected components with the new point. In order to store connected components, you may use DSU.

Now, let's handle areas. Consider a face with vertices a, b and c. If the face is horizontal, we will add its area to the area of a component when the sweep plane reaches it. Otherwise, suppose $z_a \leq z_b \leq z_c$ and the current high of the sweep plane is s. Then the area of a part of the face that is higher than the sweep plane is:

$$area(s) = \frac{\int_{z=s}^{z_c} l(z)\, dz}{\sin(\alpha)}$$

Where α is an angle between the plane of the face and a horizontal plane. And $l(z)$ is the length of a section of the face by a horizontal plane on high z. Function $l(z)$ is linear on segments $[z_a, z_b]$ and $[z_b, z_c]$. So, function $area(s)$ is a quadratic on the same segments. So, we can maintain the sum of such quadratic functions over all faces of one connected component. And when we meet a new point, we iterate over all faces that contain this point and modify the quadratic function for that face.

Finally, when the sweep plane reaches the high of some query, we look at the connected component of the point in the query.

Problem H. Heroes of Might

Problem author: Nikolay Budin; problem developer: Borys Minaiev

The full editorial of this problem could be pretty long, so we just provide some key ideas required for the solution without going into much detail.

Let's first discuss the slow solution, which works when the total number of rounds is not big. For each group of peasants, we can generate an array of integers k_i — number of peasants killed on i-th attack of this group. For example if damage is equal to 15, the health of each peasant is 10, and there is a group of 4 peasants, the corresponding array is equal to $[1, 2, 1]$.

After constructing an array for each group, we can "merge" them into one big array in a way which preserves the initial ordering of elements in each array. Such a merged array corresponds to some ordering in which dragon attacks groups. Optimal merging should maximize the sum of all prefix sums of the generated array.

How to merge arrays? Let's split each array into several continuous intervals, and for each of them calculate a_i — sum of elements in the segment divided by the length of the segment. Over all possible splits we choose one with special properties:

- $a_1 > a_2 > ... > a_n$
- $(a_1, a_2, ..., a_n)$ — lexicographically largest

Such a split could be computed greedily. First, find the prefix with largest a_1, cut it, recursively find other prefixes.

There is also a nice geometrical interpretation of such a split. We can draw points $(i, \sum_{j \le i} k_j)$, and find the upper convex hull of them. Each segment of the convex hull corresponds to the segment in the optimal array split.

It could be proven that each segment of the split could stay continuous inside optimally merged arrays. Moreover, to determine the order in which segments for different groups should be merged, we can just sort them by a_i.

It also could be proven that the optimal split has $O(log)$ segments.

The only question left is how to build such segments for larger constraints. We need to find a convex hull of points $(i, \lfloor \frac{d \cdot i}{h_p} \rfloor)$. There are different possible approaches. One of the easiest is to use continued fractions. There is a detailed description of the algorithm for finding the convex hull of lattice points under the line in https://cp-algorithms.com/algebra/continued-fractions.html.

We also need to handle the last point carefully as it doesn't follow formula $\lfloor \frac{d \cdot i}{h_p} \rfloor$.

Problem I. Interactive Treasure Hunt

Problem author and developer: Pavel Marvin

Let's notice that pairs of points $(x_1, y_1), (x_2, y_2)$ and $(x_1, y_2), (x_2, y_1)$ give the same results for all possible `SCAN` requests. So we will assume that $x_1 \le x_2$ and $y_1 \le y_2$, and then check both these pairs using three `DIG` requests.

First, let's make `SCAN`s in points $(1, 1)$ and $(1, m)$.

$$A = \texttt{SCAN}(1, 1) = (x_1 - 1) + (x_2 - 1) + (y_1 - 1) + (y_2 - 1) \quad (1)$$
$$B = \texttt{SCAN}(1, m) = (x_1 - 1) + (x_2 - 1) + (m - y_1) + (m - y_2) \quad (2)$$

From these values we can find sums:

$$S_x = x_1 + x_2 = \frac{A + B + 6 - 2m}{2} \quad (3)$$
$$S_y = y_1 + y_2 = \frac{A - B + 2 + 2m}{2} \quad (4)$$

Now let's make SCAN in points $(\lfloor \frac{Sx}{2} \rfloor, 1)$ and $(1, \lfloor \frac{Sy}{2} \rfloor)$.

$$C = \texttt{SCAN}(\lfloor \frac{Sx}{2} \rfloor, 1) = (x_2 - x_1) + (y_1 - 1) + (y_2 - 1) \tag{5}$$

$$D = \texttt{SCAN}(1, \lfloor \frac{Sy}{2} \rfloor) = (x_1 - 1) + (x_2 - 1) + (y_2 - y_1) \tag{6}$$

From these values we can find the differences:

$$D_x = x_2 - x_1 = C - S_y + 2 \tag{7}$$

$$D_y = y_2 - y_1 = D - S_x + 2 \tag{8}$$

Now we can find values

$$x_1 = \frac{S_x - D_x}{2} \tag{9}$$

$$x_2 = \frac{S_x + D_x}{2} \tag{10}$$

$$y_1 = \frac{S_y - D_y}{2} \tag{11}$$

$$y_2 = \frac{S_y + D_y}{2} \tag{12}$$

Finally, we can `DIG` in cell (x_1, y_1). If we find the treasure, then the second one must be in cell (x_2, y_2). If not, then the treasures are in the cells (x_1, y_2) and (x_2, y_1).

Problem J. Job Lookup

Problem author: Vitaly Aksenov; problem developer: Mikhail Dvorkin

This problem can be solved by dynamic programming "over subsegments".

Consider a subproblem for a segment of people $[i, j]$. We want to arrange them as a subtree of the global hierarchy tree in the optimal way. Let's denote a_{ij} to be the minimal possible communication cost induced by the edges in this subtree. That is, each communication path that costs $c_{uv} \cdot d_{uv}$ can be fragmented as the sum of d_{uv} payments of size c_{uv} being paid in each edge of the path. So in our dynamic programming value a_{ij} we will only consider the part of communication cost that is paid in the edges of the constructed subtree.

To calculate a_{ij} simply iterate over all possible root candidates $k \in [i, j]$. For a specific k the communication cost in the subtree $[i, j]$ is composed of: $a_{i,k-1}$, $a_{k+1,j}$, the communication cost paid in the edge from k to its left child (if any), and the communication cost paid in the edge from k to its right child (if any).

The communication cost paid in the edge from k to its left child is the sum of c_{uv} over all pairs (u, v) such that u is in $[i, k-1]$ and v is not (indeed, these are all pairs of people that do pay in this edge). This value is simply a sum of two subrectangles in the matrix c. The same obviously applies to the cost in the edge to the right child.

If the prefix sums (or a similar data structure) is precalculated for the matrix c, the cost for specific k can be calculated in $O(1)$ time, the value of a_{ij} — in linear time, and the entire problem — in cubic time.

Some information about the origin and the relevance of the problem can be found in the SplayNet paper, section III A: `https://www.univie.ac.at/ct/stefan/ton15splay.pdf`

Problem K. Kingdom Partition

Problem author: Maxim Akhmedov; problem developer: Niyaz Nigmatullin

Let us start by writing down a matrix of coefficients applied to the cost of an edge in the resulting functional depending on possible part belonging of its endpoints:

	A	B	C
A	2	0	1
B	0	2	1
C	1	1	0

Table 1. Desired coefficient matrix

The shape of the problem hints that it is somehow related to minimum cut, but the standard minimum cut-driven technique expresses the division of vertices into two parts, while we are asked about dividing vertices into three parts A, B and C. The main trick is to perform a frequently appearing skew-symmetric transformation of a graph. Replace each vertex v with two vertices v_1 and v_2 and each edge uv of cost l with two edges u_1v_2 and u_2v_1 of the same cost l.

Consider an arbitrary cut (S, T) of a new graph into two disjoint vertex sets S and T. Each vertex v of the original graph may be seen as being in one of four states SS, ST, TS and TT depending on whether each of v_1 and v_2 belongs to S or T. Write down a similar matrix of possible values of the coefficient applied to the cost of an edge (of the original graph) in the cut value $cut(S, T)$:

	ST	TS	SS	TT
ST	2	0	1	1
TS	0	2	1	1
SS	1	1	0	2
TT	1	1	2	0

Table 2. Coefficient matrix from cuts in a skew-symmetric graph

Note that the desired matrix is a submatrix of the matrix above. This "coincidence" hints that we must restrict vertex a to be an ST-vertex and vertex b to be a TS-vertex. Note that this can be done by connecting a source vertex s with a_1 and b_2, and connecting a_2 and b_1 with a sink vertex t using edges of infinite capacity and considering $s - t$ cuts in the resulting graph.

The last remaining issue is that we have distinct classes of SS and TT vertices. It turns out that the minimum cut may always be chosen such that there are no TT vertices; indeed, make all TT vertices be SS vertices. As a result, no edge coefficient would increase; moreover, some $SS - TT$ edge coefficients would become $SS - SS$ edges, decreasing their coefficients from 2 to 0[1].

Combining everything together, we get a solution that constructs a skew-symmetric graph, finds a minimum cut in it using any appropriate maximum flow algorithm (e.g. Dinic algorithm), and then recovers the desired partition as $A = ST$, $B = TS$ and $C = SS \cup TT$.

[1]An educational remark. The last result is a special case of a generic *cut function submodularity* property: if $cut(X)$ is a value of the cut between X and $V \setminus X$, then

$$cut(X \cap Y) + cut(X \cup Y) \le cut(X) + cut(Y).$$

Now apply this property to $X := S$ and $Y := T'$ where T' is a set of vertices symmetric to the vertices in T (i.e. $v'_1 = v_2$ and $v'_2 = v_1$), and obtain the previous result.

Problem L. Labyrinth

Problem author: Michael Mirzayanov; problem developers: Sergey Melnikov, Michael Mirzayanov

Let the required two paths have the form:

- $s = u_1, u_2, \ldots, u_x, t$;
- $s = v_1, v_2, \ldots, v_y, t$.

Let us show that there is always a pair of required paths such that the vertices u_x and v_y (that is, the penultimate vertices of the paths) lie in different subtrees of any depth-first search tree rooted at s. This only applies when both vertices are different from s.

There is a separate corner case in this problem when $u_x = s$ or $v_y = s$. Just remember about it, it is easy to handle it in code.

Indeed, let's take t such that the distance from s to t is minimal.

Suppose this is not the case and there is a depth-first search tree such that vertices u_x and v_y are in the same DFS subtree rooted at s. But since t is the answer, there are two distinct vertex-disjoint (except vertices s and t) paths: $u_1, u_2, \ldots, u_x, t$ and $v_1, v_2, \ldots, v_y, t$.

Since $u_1 \neq v_1$, then at least one of these paths starts not in the subtree where u_x and v_y are located. Without loss of generality, let this path be $u_1, u_2, \ldots, u_x, t$. Find the first vertex in it (minimum index j) such that u_j belongs to the path in the DFS tree from s to u_x. Thus, we have built a pair of non-intersecting paths (from s to u_j) that end at the same vertex, and this vertex is closer to s than t. We get a contradiction with the fact that the distance from s to t is minimal.

Thus, it is enough to run a depth-first search and choose such a vertex as t, such that:

- let the DFS parent of vertex t be vertex u_x,
- t has an edge from some vertex v_y, which in this DFS tree is in a different DFS subtree than t relative to the root s (or $v_y = s$ and $u_x \neq s$).

These paths in DFS tree (from s to u_x and from s to v_y) will induce the required paths.