Problem A. Almost Balanced Tree

Problem author and developer: Pavel Mavrin

The key observation is that it's easy to build an almost balanced tree if you have only nodes of weight 1, but it's almost always impossible if you have only nodes of weight 2.

If you develop this idea further, you can prove the following fact: if it is possible to build an almost balanced tree for (a, b), it's also possible to build it for $(a + 2, b - 1)$, i.e. you can replace one node of weight 2 with two nodes of weight 1.

Now we can calculate value $d[s]$, the minimal number of nodes of weight 1 required to be able to build an almost balanced tree with a total weight of all nodes s.

This value is recalculated easily: you need to pick some node as root, its weight is 1 or 2. When you fix its weight, you know the weights of both subtrees, because you know their sum and they have difference at most 1. So if root has weight 1, then the minimal total number of 1 is $1 + d[\lfloor \frac{s-1}{2} \rfloor] + d[\lceil \frac{s-1}{2} \rceil]$, and if it is 2, then the minimal total number of 1 is $d[\lfloor \frac{s-2}{2} \rfloor] + d[\lceil \frac{s-2}{2} \rceil]$.

Now we can check that we have enough nodes of weight 1, and if it so, build the tree recursively.

Problem B. Brain-teaser

Problem author and developer: Roman Elizarov

The problem is solved with a brute-force search. However, a naive search that checks all 10! possible assignments of digits to letters for each word in the dictionary will not fit into the time limit, so a more efficient way to arrange the search is needed. There are many approaches to get a solution that fits into the time limit. One of them is described below.

Create a *trie* of reversed dictionary words. Do a recursive exhaustive search from the last digits of the given equation. For example, in the case of `SEND+MORE` start with the assignment of letter `D` (the last letter of `SEND`) to all the digits form `0` to `9`, followed by the assignment of letter `E` (the last letter of `MORE`). Now, the digit at the last position of the 3rd word is known and you can use the trie to assign it to at most 26 of all the possible last letters of the words from the dictionary. Recursively move to the second-to-last letter, taking into account the carry from the last digit and moving to the corresponding child node in the trie, etc. When moving recursively, the letters that are already assigned do not need to be exhaustively checked again, and the digits that were already used cannot be assigned to a different letter.

Whenever all the letters are consistently assigned to digits, increment the number of solutions for the corresponding dictionary word in the current node of a trie.

Mind the restriction that the first digit of a number cannot be zero. In the `SEND+MORE` example, the letters `S` and `M` can only be assigned digits from `1` to `9`.

At the completion of this exhaustive search, you've counted the number of brain-teaser solutions for each dictionary word. The answer is then all the words with one solution.

Problem C. Color the Tree

Problem author: Andrew Stankevich; Problem developer: Borys Minaiev

Let's solve the task recursively. To build the solution for some node, we need to build solutions for all children, join them, and add the current vertex. By the solution for a node we mean a list of states, where each state represents a color of each node in the subtree. Obviously, two neighbor states should have exactly one vertex with different colors. Also, all states of the solution should be unique.

Solution for the leaf is just two states — this vertex colored red and green.

Let's say we have solutions for two children represented by lists of states s_1, s_2, ..., s_n and t_1, t_2, ... t_m. We can build a solution for two subtrees by merging the lists this way:

- (s_1, t_1)
- (s_1, t_2)
- ...
- (s_1, t_m)
- (s_2, t_m)
- (s_2, t_{m-1})
- ...
- (s_2, t_1)
- (s_3, t_1)
- (s_3, t_2)
- ...
- (s_n, t_m) or (s_n, t_1)

You can check that this list satisfies the requirements to be the solution for two nodes. In a similar way, all children of the node could be merged.

To add the current vertex to the solution of all children, we can simply add the current vertex colored red to all states, and also prepend one additional state with all vertices colored green.

It is easy to prove that solution constructed this way gives the longest possible sequence of operations because it iterates over all possible valid colorings of the tree.

Problem D. Down We Dig

Problem author and developer: Mikhail Dvorkin

Consider the game played on some specific afternoon. This is a "classical" game theory problem: some steps are winning positions (player who starts at this step wins the game), some steps are losing positions (player who starts at this step loses the game), and the rules for finding which ones are which are:

If there is a possible move from some position to a losing one, then this position is winning. Otherwise — if all moves lead to winning positions — then this position is losing.

With these rules you can mark winning/losing positions from the bottom to the top... but that would take $O(n^2)$ time for all afternoons! So let's introduce "memoization" to this procedure.

Note that according to the rules of the game you can only move at most 8 steps downwards in one move. And therefore, when calculating whether some step is winning or losing, the only information you really need to know is the "winningness" of the 8 steps below this one.

So if at some point in the past you have already examined some steps with some bitmask of "winningness" of the lower 8 steps, you don't need to make these calculations again, but rather take the stored result from the memoization table.

This way, you will do at most $n \cdot 2^8$ examinations of positions, each of which consists of 8 lookups to some lower steps. The formal work time is $O(n \cdot 2^m \cdot m)$, where $m = 8$ is the width of the staircase.

Problem E. Easy Measurements

Problem author: Elena Kryuchkova; Problem developer: Evgeniy Kapun

The problem asks to find the number of solutions to $a/b + c/d = b/d$ such that a and c are positive integers. Let $g = \gcd(b, d)$. The equation is equivalent to $(d/g)a = (b/g)(b - c)$. Because d/g and b/g are coprime, a must be equal to $(b/g)k$ for some integer k, and then $c = b - (d/g)k$. From $a, c > 0$ it follows that $0 < k < b/(d/g)$, so the number of solutions is $\lfloor (b-1)/(d/g) \rfloor$.

Problem F. Find a Square

Problem author and developer: Pavel Kunyavskiy

To solve the problem, one need to find for each prime number q maximal degree d_q, such that product is divisible by q^{d_p}, and calculate $\prod q^{d_q-(d_q \bmod 2)}$. The result does not depend on primes for which d_q is equal to 1, so we can ignore them, and need only to find all primes for which d_q is at least 2.

Given a prime number q, there can be at most 2 reminders modulo q, such that $p(x) \equiv 0 \pmod q$, if $gcd(q, 2a) = 1$. There are at most $O(\log a)$ primes which are not co-prime to $2a$. So one can just use trial division for all these primes and for all numbers. This part will work in $O(n \times \log a)$ time. For all other primes, we need to solve the quadratic equation in the field of reminders and use trial division only for numbers with good reminders. If Q is set of primes to test, this will work in $O(\sum \left\lceil \frac{n}{p} \right\rceil) = O(n \log M + |Q|)$ total, which is fast enough, if $|Q|$ is small enough.

The only two remaining parts are solving the quadratic equations and finding primes, which would have at least one solution, less than n.

To solve the quadratic equation, one can use well-known formula $x = \frac{-b \pm \sqrt{b^2-4ac}}{2a}$. If no square root exists in the field of reminders, there is no solution. To find the square root itself, one can use *Tonelli–Shanks algorithm* or *Cipolla algorithm* (both can be found in Wikipedia) or any other algorithm that is fast enough. Both of them work in $O(\log q)$ time on average, so total time for all primes would be $|Q| \log M$.

How to find good set $|Q|$? First of all, let's test all primes less than cubic root of the maximal value. Ant let's test all remaining numbers if they are perfect square. Now, all remaining integers are either primes or product of two distinct big primes. So, we are now interested only in primes which can be divisors of at least two integers. Consider $gcd(a \cdot x^2 + b \cdot x + c, a \cdot y^2 + b \cdot y + c)$. By subtracting one from another, it's equal to $gcd(a \cdot x^2 + b \cdot x + c, a \cdot (y^2 - x^2) + b \cdot (y - x)) = gcd(a \cdot x^2 + b \cdot x + c, (y - x) \cdot (a \cdot (y + x) + b))$. So, if prime is divisor of at least two of remaining numbers, it should be either divisor of number not greater then n, or number of form $a \cdot x + b$ with x not greater then $2 \cdot n$.

All divisors of numbers $a \cdot x + b$ for many first x can be found using same idea. For each prime number q there at most 1 reminder, for which $a \cdot x + b \equiv 0 \pmod q$. So we can find it, and make a trial division for all $\frac{2n}{q}$ possible numbers. We need to test all primes less then $\sqrt{a \cdot (2n) + b}$, and all remaining numbers would be primes. In total, size of set $|Q|$ would be at most $\pi(max(\sqrt[3]{a \cdot n^2 + bn + c}, 2n, \sqrt{an + c})) + 2n$, where $\pi(x)$ is number of primes less than x, which is less then 2 million primes to test.

Problem G. Geometrical Combinatorics

Problem author and developer: Pavel Kunyavskiy

It is well known, that k-th number in n-th row of a Pascal's triangle is equal to $\binom{n}{k} = \frac{n!}{k!(n-k)!}$. Unfortunately, there can be $O(C^2)$ integers inside the triangle's intersection. So we need to split the triangle's intersection into several convenient parts. There two intuitive ways to split it — to parts with fixed n and parts with fixed k. Both of them lead to success.

Parts with fixed n are horizontal lines, so one can check all horizontal lines and intersect them with the given triangle. The same can be done with diagonals to find all parts with fixed k.

To sum the numbers with fixed k, identity $\sum_{n=0}^{r} \binom{n}{k} = \binom{r+1}{k+1}$ can be used, to represent the sum of one part as $\binom{r+1}{k+1} - \binom{l}{k+1}$.

For parts with fixed n there is no such a good small form for the sum on one raw. But, one has already calculated sum for previous raw. $\sum_{k=l}^{r} \binom{n}{k} = \sum_{k=l}^{r} \binom{n-1}{k-1} + \binom{n-1}{k} = 2 \sum_{k=l}^{r} \binom{n-1}{k} + \binom{n-1}{r} - \binom{n-1}{l-1}$.

To calculate $\binom{n}{k}$ fast, one can precalculate all factorials and factorial inverses, and just multiply three numbers.

Problem H. Hit the Hay

Problem author and developer: Petr Mitrichev

At any moment of time, we can be in one of 5 states: baby state 0, baby state 1 + we're asleep, baby state 2 + we're asleep, baby state 1 + we're not asleep (let's call this state 3), baby state 2 + we're not asleep (let's call this state 4). So if the problem was dicrete (there was some unit of time), we'd use straightforward dynamic programming "given that we're in one of those 5 states, and there are t hours remaining before the alarm, what is the maximum expected amount of sleep we can get?". Let's denote these values by $e_i(t)$, where i runs from 0 to 4.

These values still make sense in the continuous problem, but we can no longer use dynamic programming to find them since we have an infinite number of moments of time. Instead, we can write for ϵ that tends to 0, for example for e_1:

$$e_1(t+\epsilon) = \epsilon + p_1^\epsilon \cdot e_1(t) + (1 - p_1^\epsilon) \cdot (q_0 \cdot e_0(t) + (1 - q_0) \cdot e_2(t))$$

By subtracting $e_1(t)$ and dividing by ϵ as it tends to 0, we get:

$$e_1'(t) = 1 + \log p_1 \cdot e_1(t) - \log p_1 \cdot (q_0 \cdot e_0(t) + (1 - q_0) \cdot e_2(t))$$

So we have five such differential equations, with the additional complexity that for states 3 and 4 we can make a choice: we can go to sleep and transition to state 1 or 2 correspondingly. Essentially, we have:

$$e_3(t+\epsilon) = \max(e_1(t+\epsilon), ...)$$

When t is small, this max will always be resolved to the first branch, because the differential equation for e_1 has "1+", which dominates the rest when t is small. So we can find a solution to the system of three differential equations for small t, and then we need to find the moment when it becomes advantageous to select the second branch in the max for e_3: it happens when

$$-\log p_1 \cdot q_0 \cdot (e_1(t) - e_0(t)) \geq 1$$

The value of $e_1(t) - e_0(t)$ will be growing monotonically (see below for proof), therefore we can use binary search to find this moment. It will always be optimal to choose the second branch in the max() after this point (since with more time available we get even more benefit from preventing the baby from transitioning from state 1 to state 0), so we get a new system of four differential equations that we need to solve.

Finally, it turns out that for e_4 it is always advantageous to go to sleep (it's not clear what we'd be waiting for if the baby is in the best possible state), so we have no other cases to consider.

The arguments above are a bit handwavy, so to be extra sure we have also verified the various assumptions here (usability of binary search to find transition point, the fact that after that transition point we should never go to sleep in state 1, the fact that for e_4 we should always use the first branch) numerically, by checking all valid combinations of p_0, p_1, p_2 and q_0 and many moments in time for each combination.

How to solve the two systems of differential equations? For a homogenous system (without "1+" on the right-hand side) of the form $e' = A \cdot e$ (where A is a matrix), the solution is given by $e(t) = e^{A \cdot t} \cdot e(0)$, where the exponent of a matrix can be found using the Taylor series. To improve precision in this computation when the matrix has large values, we can compute $e^{A \cdot t} = (e^{\frac{1}{k} A \cdot t})^k$.

To solve the non-homogenous system, we need to find any solution of the non-homogenous system and then add to it all solutions of the homogenous system. In this case 0 will be one of the eigenvalues of matrix A (because the sum of each row is 0), so a solution for the non-homogenous system can be found in the form $e = p \cdot t + q$, where p and q are constant vectors. Alternatively, we can add an extra variable with a derivative of 0 to transform our system to a homogenous one.

We can also use the Runge–Kutta fourth order method to solve the system numerically, the step size of $\frac{k}{200}$ seem to be enough to reach the required precision under our constraints.

Now, let's prove that the decision we binary search on is monotonic. We will use the following lemma:

Lemma. Suppose we have two probability distibutions on the baby's states r_0, r_1, r_2 ($\sum_i(r_i) = 1$) and s_0, s_1, s_2 ($\sum_i(s_i) = 1$) such that $r_0 \geq s_0$ and $r_2 \leq s_2$. The two distributions correspond to two different scenarios. Then, after any positive amount of time t passes with the baby states changing according to the rules from this problem (without the parent preventing any transitions) in each scenario, the new probability distributions r'_i and s'_i will still satisfy $r'_0 \geq s'_0$ and $r'_2 \leq s'_2$.

Consider the moment where these constraints might've been violated. At this moment, one or two of those constraints must be an equality. If both of them are an equality, then we have two identical probability distributions, and therefore they will stay identical as well. If only one of them is an equality, say, $r_0 = s_0$, then we still have $r_2 < s_2$, and therefore $r_1 > s_1$, and therefore more probability will flow from state 1 to state 0, so we will get $r_0 > s_0$ again after a small amount of time. A symmetric argument applies for the $r_2 = s_2$ case.

Because of the lemma, if we start in state 0 and spend some time, our probability to be in state 0 will always be higher compared to the case where we started in state 1 and spent the same amount of time. Since $e_0(t)$ is the integral of one minus that probability when we start in state 0, and $e_1(t)$ is the integral of one minus that probability when we start in state 1, this shows that $e_1(t) - e_0(t)$ is monotonically increasing for the case where the parent does not interfere.

Problem I. Interactive Knockout

Problem author and developer: Ilya Zban

There are a lot of ways about how to solve this problem.

One of the easiest ways to solve this is to investigate the patterns of random moving participants. It turns out that without the opponent such a player loses pretty fast on a field with $n = 20$ — if we take the total number of the cells on the field as s, then on average they lose after $\frac{s}{20}$ turns, and very probably (in 10^5 tries) they would lose after $\frac{s}{3}$ turns. So, we can try to claim $\frac{s}{3}$ of the field and hope that the opponent loses before us.

We can claim one of the parallelograms $A = \{(0,0),(0,n),(-n,n),(-n,0)\}$ or $B = \{(0,0),(n,-n),(0,-n),(-n,0)\}$ with high probability — first, we go to the $(0,0)$, and then if enemy has negative y-coordinate we try to claim parallelogram A and B otherwise. We will always be the first to get to $(0,0)$ and $(0,n)$ or $(n,-n)$, but sometimes jury can interrupt us by randomly going in our parallelogram through segment $(-n,0)\ldots(-\frac{n}{2}-1,0)$. This solution works almost fine, but we need to adapt our strategy to this case.

Again, there are multiple ways to fix this problem. One of them — when we go from $(-n,0)$ to $(-\frac{n}{2}-1,0)$, we can check, if the opponent is inside of the parallelogram, close the figure, and go to other connected component — we won't interact with the jury, and we can either win by going over $\frac{s}{3}$ cells in a parallelogram, or we can stand outside, which has even more space. It would be a little harder to find a long path there, but it is still easy to win. This solution fails with a probability close to $\frac{1}{15\,000}\cdots\frac{1}{10\,000}$, which is enough to pass.

Another way is to try to claim a smaller parallelogram if we see that the opponent is close. In other words, for the case A we go from $(0,n)$ not to $(-n,n)$, but to some $(-k,n)$ $(0 \leq k \leq n)$, such that the jury won't interrupt us. This solution works even better and fails with probability close to $\frac{1}{100\,000}$.

Problem J. Jumping Cat

Problem author and developer: Gennady Korotkevich

This is essentially the shortest path problem with a geometry flavor.

It can be seen that the cat can't avoid building corners in its trajectory for too long: in particular, if the cat jumps down, it has to start from a corner, and if it jumps up, it has to land at a corner.

We can build a graph where building corners are the only vertices, and edges correspond to trajectory parts between them. One such part could take any of the following forms:

1. a direct jump or a direct walk;
2. walk to the right, and then jump up;
3. jump down, and then walk to the right;
4. jump down and immediately jump up (as in Example 1);
5. jump down, walk to the right, and then jump up.

In cases 2 and 3, the jump has to have a length of exactly L in the shortest trajectory.

In case 4, the shortest path should satisfy the law of reflection if possible. It might be impossible due to any of the jumps being too long: e.g., if the first jump (going down) is too long, we have to move the "reflection" point to the left to make the first jump have the length of exactly L. If the second jump becomes too long after that, we have to resort to case 5 and walk in-between jumps.

The graph built this way is directed acyclic (all edges go from left to right), hence we can find the shortest path using simple dynamic programming.

Alternatively, we can build the graph on vertices that are the building corners, optimal reflection points, and all intersections of circles with centers in the building corners and radius L with surface segments, and run any shortest path algorithm.

Problem K. Kate's 2021 Celebration

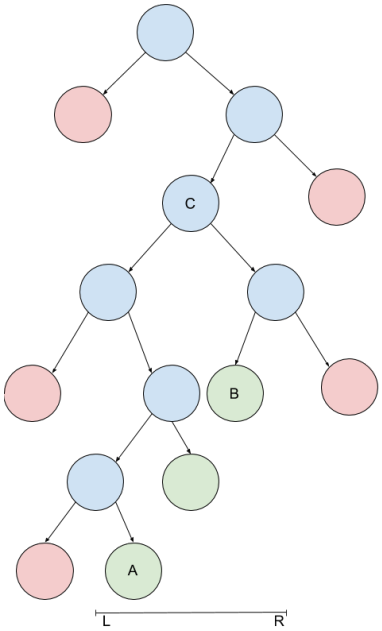
Problem author: Petr Mitrichev; Problem developer: Roman Elizarov

In this problem, we need to check that a given string of digits contains all the digits that are needed to get 2021. For that, you shall count the number of occurrences of digits `0`, `1`, and `2`. Then check that `0` occurs at least once, `1` occurs at least once, and `2` occurs at least twice. That gives you all the packs that Kate can buy to get digits for her 2021 celebration. Now the final task is to select the pack with the smallest price among those and output its index.

Problem L. Lookup Performance

Problem author: Vitaly Aksenov; Problem developer: Niyaz Nigmatullin; Tutorial: Vitaly Aksenov, Niyaz Nigmatullin

Let us see which nodes the request $[L, R]$ visits. Please, see the picture of calls: the intervals of green nodes lie inside $[L, R]$, the intervals of blue nodes intersect $[L, R]$, and the intervals of red nodes do not intersect $[L, R]$. The null-nodes can represent both green nodes and red nodes, we can also assign each those nodes an interval (that doesn't contain any key from the set).



We start at the top of the tree. We are interested in the nodes which interval intersects $[L, R]$ but not belongs to it. At first, $[L, R]$ lies inside the interval of a node and fully lies inside the interval of only one child (the neighboring call is red and finishes simultaneously since the corresponding interval does not intersect $[L, R]$). Then, at some node C, $[L, R]$ lies in the interval of C and intersects the intervals of both children: both calls of `lookup` look for the nodes in both children. After this node, `lookup` calls follow the path to some node in the subtree of the left child and the path to some node in the subtree of the right child. What are these target nodes? They are the parent of a node A (probably null) with the leftmost interval that lies in $[L, R]$ and the parent of a node B (probably null) with the rightmost interval that lies in $[L, R]$. Since A is the leftmost green node, the key of the parent of A is either the maximum key less than L, or the minimum key greater or equal to L. Otherwise, the left child of the parent of A would call its children, too. These two values ($\max S$ less than L and $\min S$ not less than L) are in such nodes, that one of them is the ancestor of the other. When we reach any of those two nodes, its left child is red, and its right child is green. So we as the left target we choose among $\max S$ less than L and $\min S$ not less than L the one that is an ancestor of the other. The same can be done for R. Then vertex C is just the LCA (the least common ancestor) of those target nodes.

To find the target nodes, we need a sorted array of all the nodes ordered by their keys and binary search to find two values. For the following, we need some LCA solving algorithm and compute the depths of the nodes. Then the answer is $2 \cdot (d_{t_L} + d_{t_R} - d_{\mathrm{LCA}(t_L, t_R)}) + 1$, where d_v is the depth of node v such, that the depth of the root being equal to 1, and t_L and t_R are the left and the right target nodes.

You have to also consider corner cases, where L is less than equal to all the keys, and where R is greater or equal to all the keys, or both.

Problem M. Miser

Problem author and developer: Vitaly Aksenov

Let us build a directed graph on the days as vertices: there exists an edge from day i to day j if $i > j$ and there is a person that comes on both of these days. It can be seen that the answer to the problem is the length, in vertices, of the maximal path in this directed graph.

Now, we simply iterate through the days from the last to the first one and calculate what is the length of the maximal path starting from the corresponding day. For that, we maintain for each person p the length `l[p]` of the maximal path starting from some later day, when p visited the University. When we get to the day d we iterate through all the people that visited the University that day and find the maximal `l[p]`,

then the length of the maximal path from this day is the found maximum plus one. Finally, we update `l[p]` for each person that comes at day d.

Problem N. New Flat

Problem author: Georgiy Korneev; Problem developer: Egor Kulikov

Solution outline:

- Note that if we've tranformed AB into some vector v we can then move it to any position within polygon.
- It can be proven that any reachable segment is reachable by a series of moves and rotations, and we only need to consider rotations where one of segment ends is fixed in vertex of the polygon.
- We will define equivalency relation on hordes of the polygon (edges and diagonals). Two hordes are equivalent if we can transform segment lying on one of this hordes to one lying on the other (hordes that are shorter than segment are only equivalent to themselves).
- We will consider triples of vertices U, V and T such that they are all distinct and V and T are consecutive vertices and $|UV| \geq |AB|$ and $|UT| \geq |AB|$. If distance from U to any point on segment VT is at least $|AB|$ then UV is equivalent to UT and VU is equivalent to TU. It can be proven that our equivalency relation is a closure of all such relations.
- If all hordes longer than $|AB|$ are equivalent to each other then we can freely rotate segment within polygon and answer is $90°$.
- Otherwise we will find vertex such that AB can be moved so one of its ends lies in this vertex. Then we will rotate it to one of the neighboring hordes (at least one of them should be reachable). The we will consider all hordes equivalent to this one and will try to rotate segment within neighboring triangles and calculate answer as maximal of angles between corresponding vector and AB. Note that we will need to minimize answer to $90°$ if it happen to be bigger.

Problem O. Optimum Server Location

Problem author and developer: Maxim Ahmedov

The problem may be formulated as follows. Given $a_k \in \mathbb{R}, c_{ik} \in \mathbb{R}_+, d_{ij} \in \mathbb{R}_+$ (hereinafter $1 \leq i, j \leq n$, $1 \leq k \leq m$):

$$\begin{cases} x_i \in \mathbb{R} \\ \sum\limits_{i<j} |x_i - x_j| d_{ij} + \sum\limits_{i,k} |x_i - a_k| c_{ij} \to \min \end{cases} \tag{1}$$

Let's transform this (convex) problem into linear programming problem. Let's express $x_i - x_j$ as $r_{ij}^+ - r_{ij}^-$ where $r_{ij}^+, r_{ij}^- \in \mathbb{R}_+$ and replace $|x_i - x_j|$ with $r_{ij}^+ + r_{ij}^-$ in objective function. Note that such substitution is not uniquely defined, but $r_{ij}^+ + r_{ij}^-$ belongs to the objective function with positive coefficient, so the optimum is attained when at most one of r_{ij}^+, r_{ij}^- is non-zero. It is easy to see that non-zero variable will be equal to the absolute value of $x_i - x_j$.

Perform the same substitution for $|x_i - a_k|$ replacing it with $s_{ik}^+ + s_{ik}^-$.

The problem now looks like the following:

$$\begin{cases} x_i \in \mathbb{R} \\ r_{ij}^+, r_{ij}^-, s_{ik}^+, s_{ik}^- \in \mathbb{R}_+ \\ x_i - x_j - r_{ij}^+ + r_{ij}^- = 0 \quad \forall i < j \\ x_i - s_{ik}^+ + s_{ik}^- = a_k \quad \forall i, k \\ \sum\limits_{i<j} d_{ij} r_{ij}^+ + \sum\limits_{i<j} d_{ij} r_{ij}^- + \sum\limits_{i,k} c_{ik} s_{ik}^+ + \sum\limits_{i,k} c_{ik} s_{ik}^- \to \min \end{cases} \tag{2}$$

Consider the dual linear programming problem. Variables α_{ij} and β_{ik} correspond to the equations in the problem above, thus they are unconstrained in sign.

$$\begin{cases} \alpha_{ij}, \beta_{ik} \in \mathbb{R} \\ -\alpha_{ij} \le d_{ij} \quad \forall i < j \\ \alpha_{ij} \le d_{ij} \quad \forall i < j \\ -\beta_{ik} \le c_{ik} \quad \forall i, k \\ \beta_{ik} \le c_{ik} \quad \forall i, k \\ \sum\limits_{i<j} \alpha_{ij} - \sum\limits_{j<i} \alpha_{ij} + \sum\limits_{i,k} \beta_{ik} = 0 \forall i \\ \sum\limits_{i,k} \beta_{i,k} a_k \to \max \end{cases} \tag{3}$$

This problem may be treated as a minimum cost flow problem. Consider graph consisting of n vertices (indexed by i or j) organized as a clique and m vertices (indexed by k), each connected with all n vertices of a clique.

Consider α_{ij} $(i < j)$ to be the value of a flow going from i to j and β_{ik} to be the value of a flow going from i to k. Values d_{ij} and c_{ik} become capacities constraining flows on these vertices.

The fifth condition in the program above looks like a flow conservation condition for all vertices of a clique. Flow conservation is not required for m non-clique vertices, but it may easily be fixed by introducing the extra vertex s and connecting it to all of the m non-clique vertices. Objective function may be rewritten using the flow between s and all of these vertices making our problem to be minimum cost circulation problem.

Suppose we have solved the dual problem. The certificate for finding minimum cost circulation is a potential function π; it may be recovered as a distance function from s in residual network. It turns out that the potential function (as a dual object for the dual problem) simply defines the solution for the primal problem: simply let x_i be equal to π_i.

From the complexity point of view, there is a variety of ways how to solve the flow part of the problem. The most basic and generic cycle cancelling minimum cost circulation algorithm works in $\mathcal{O}(UVE)$ where U is the value of the cost and VE is the complexity of Ford-Bellman algorithm for finding positive cost cycle in residual network. If n stands for $\max\{n, m\}$ then $V = \mathcal{O}(n)$, $E = \mathcal{O}(n^2)$ and $U = \mathcal{O}(n^2 PQ)$ where P is the maximum value of a_i and Q is the maximum coefficient among c_{ik}, d_{ij}. Overall complexity looks like $\mathcal{O}(n^5 PQ)$ This seems too much as coordinates are pretty large.

One way to succeed is to note that the only edges with costs in the network are those adjacent to s. Consider the value of "transit" flow through the vertex s; this value increases on each cancelled cycle limiting the number of iterations with $\sum c_{ik}$ as any positive cycle would cross the cut between m and n vertices providing the better complexity $\mathcal{O}(n^5 Q)$. Also, one may get rid of Ford-Bellman replacing the positive cycle detection routine with DFS (as there are no costs for edges non-adjacent with s) leading to complexity $\mathcal{O}(n^4 Q)$ which is OK to pass this problem.

Another way is also to utilize the fact that costs are provided only for the edges adjacent to s and find the minimum cost circulation by reducing it to n instances of minimum cut problem. In terms of the original problem it looks pretty nice: one must solve $m - 1$ instances of the original problem "narrowed"to the segments between adjacent CDN points.