icpc International Collegiate Programming Contest

The 2025 ICPC Europe Championship

Solutions

host university

A Condorcet Elections

Author: Alice Sayutina
Preparation: Alice Sayutina

This problem is inspired by Condorcet's paradox. To put it simply, Condorcet's paradox states that for certain voters preferences, it's impossible to always resolve the elections fairly. No matter the outcome, more than half of the voters would prefer to change the result to some other outcome. This is illustrated by the second example in the statement.

The problem's answer is always `YES`. It's possible to construct a list of votes such that any (not trivially contradicting) list of statements "candidate x_i has defeated candidate y_i" can be satisfied.

Let $S = \{(a, b) \mid 1 \le a, b \le n, a \ne b\}$, and let $R \subseteq S$ be the list of requirements. Let $\delta_{a,b}(\text{ans})$, where $(a, b) \in S$ and "ans" is our answer, be how many votes prefer a to b minus how many votes prefer b to a. We want to ensure that $\delta_{a,b}(\text{ans}) > 0$ for every $(a, b) \in R$.

For every $(a, b) \in S$, it's actually possible to construct votes π_1, π_2, such that:

- $\delta_{a,b}(\{\pi_1, \pi_2\}) = 2$.
- $\delta_{x,y}(\{\pi_1, \pi_2\}) = 0$ for all $(x, y) \in S$, $\{x, y\} \ne \{a, b\}$.

Then clearly adding π_1, π_2 for every $(a, b) \in R$ results in a correct answer. Thus, the jury solution uses at most $2\binom{n}{2} \le n^2$ votes.

Let us construct π_1, π_2. For the sake of exposition, let us assume $a = 1$, $b = 2$. Observe that the π_1, π_2 given below satisfy the requirements. Votes π for any $(a, b) \ne (1, 2)$ can be constructed by the re-numeration.

- $\pi_1 = (1, 2, 3, 4, \ldots, n-1, n)$
- $\pi_2 = (n, n-1, \ldots, 4, 3, 1, 2)$

B Urban Planning

Author: Petr Mitrichev
Preparation: Petr Mitrichev

Solutions that do not work

First of all, let us consider a square with side n, consisting only of parks. It has $\left(\frac{n(n-1)}{2}\right)^2$ rectangular walks, since there are $\frac{n(n-1)}{2}$ ways to choose the top and bottom rows, and $\frac{n(n-1)}{2}$ ways to choose the left and right columns. We will denote this number of rectangular walks in a square park as $f(n)$.

Now the logical idea would be to find the largest $f(n)$ that does not exceed k, create a park with side n, and then try to create additional independent parks using the remaining space on the grid to achieve $k - f(n)$ rectangular walks there.

When $k = 4 \cdot 10^{12}$, we would choose $n = 2000$, get $f(2000) = 3\,996\,001\,000\,000$ walks that way, and then we would need to squeeze the remaining $3\,999\,000\,000$ walks into the rest of the 2025 times 2025 grid, which looks like two overlapping 24 times 2025 rectangles after we cut out the 2000 times 2000 rectangle plus adjacent cells to make sure the additional parks do not interact with the big one. However, a 24 times 2025 rectangle has only $565\,606\,800$ rectangular walks, so even with two of those we have no chance to reach the required number. Therefore the idea to have a completely independent large square park is too crude, and we need to improve it.

Instead of stopping at $f(n)$, let us add more park cells one by one to go from a square with side n to a square with side $n+1$, first completing column $n+1$ from top to bottom, then completing row $n+1$ from left to right. For each newly added park cell, the number of rectangular walks will increase by at most n^2. This way we can find an incomplete square with x rectangular walks such that $k - x < n^2$, so we will only need to squeeze less than n^2 additional walks in the remaining space.

In practical terms, when $k \le 4 \cdot 10^{12}$, at the first step we will get a (partial) square with side at most 2001, and have at most $3\,999\,999$ remaining walks, then at the second step we will get a square with side at most 64, at the third step at most 12, then at most 6, then at most 4, then at most 3, at which moment we would have at most 3 remaining walks, which we can address by at most 3 squares with side 2.

It is still impossible to put a square of side 2001 and a square of side 64 without overlapping into a grid with side 2025, but now it is clear that we have enough area in our grid for this, and it is only the shape that is the problem. Therefore we should use rectangles instead of squares.

Solution that works for $k \le 4 \cdot 10^{12}$

In the onsite version of the contest, the constraint was $k \le 4 \cdot 10^{12}$, for which the following approach works.

Let us start with a rectangle with height 1 and width 2025, and add a second row to it cell by cell from left to right, then the third row, and so on until we would have exceeded k rectangular walks. After this, let us skip one row and start a new rectangle with width 2025 in the same fashion for the remaining walks, and so on until we reach exactly k.

This way the first (partial) rectangle will be at most 1977 times 2025, the second at most 3 times 2025, and then at most six more 2 times 2025 ones, which fits into our grid even with the additional

empty rows between them, so this is a working solution.

Solution that works for $k \le 4.194 \cdot 10^{12}$

In the online mirror of the contest, this problem had a higher constraint: $k \le 4.194 \cdot 10^{12}$. Note that $f(2024) \approx 4.191 \cdot 10^{12}$ and $f(2025) \approx 4.2 \cdot 10^{12}$, meaning that we need a grid almost full of parks to achieve such high values of k, and we cannot afford to have several connected components.

First, we do the same as above: let us find a partial rectangle that has almost k rectangular walks. Additionally, we will force the partial rectangle to have n rows and $n+1$ columns for some value of n, and we force the removed cells (which make the rectangle partial) to all be in one corner. In other words, we start with a full n times $n+1$ rectangle, and then remove cells in the following order: first $(0,0)$, then $(0,1)$, then $(1,0)$, then $(1,1)$, then the cells with coordinates ≤ 2, and so on until the number of rectangular walks becomes $\le k$.

Since removing each particular cell reduces the number of rectangular walks by at most $n(n-1)$, we will be able to find a partial rectangle with such number of rectangular walks x that $0 \le k - x < n(n-1)$. However, for reasons that will become apparent soon, let us remove a few more cells to achieve $n(n-1) \le k - x < 2n(n-1)$.

Since $f(n+1) - f(n) = O(n^3)$, and removing a cell reduces the number of rectangular walks by $\Omega(n^2)$, we will only remove $O(n)$ cells from the full rectangle to achieve this. And because of the order in which we remove the cells, all removed cells will be concentrated in a $O(\sqrt{n})$ times $O(\sqrt{n})$ corner of the rectangle, which means that we will still have $n - O(\sqrt{n})$ completely untouched rows (with $n+1$ cells each) and $n - O(\sqrt{n})$ completely untouched columns (with n cells each).

Now we will add the remaining $k-x$ rectangular walks by adding cells to the right of the untouched rows (in column $n+2$) and to the bottom of the untouched columns (in row $n+1$), so the entire city will fit inside the $n+1$ times $n+2$ grid. If we add y consecutive cells to the right of the untouched rows, this will add $\frac{y(y-1)}{2}(n+1)$ rectangular walks, and adding y consecutive cells to the bottom of the untouched columns will add $\frac{y(y-1)}{2}n$ rectangular walks.

Now, remember that every number $z \ge n(n-1)$ can be represented as $pn + q(n+1)$ for some non-negative p and q. Since $k - x \ge n(n-1)$, we can find such non-negative p and q that $k - x = pn + q(n+1)$. And now we just need to represent p and q as the sum of values of the form $\frac{y(y-1)}{2}$, and for each such value add a corresponding segment of y extra cells to the right or to the bottom. To represent a given number p as a sum of values of the form $\frac{y(y-1)}{2}$ is a knapsack problem that we will solve greedily: take the largest value of the form $\frac{y(y-1)}{2}$ that does not exceed p, and repeat for the remainder.

Now since $k - x < 2n(n-1)$, both p and q will be $O(n)$, so the first value of y we will get will be $O(\sqrt{n})$, the second will be $O(\sqrt[4]{n})$, and so on, so the output of the greedy knapsack will fit in the $n - O(\sqrt{n})$ untouched rows/columns.

This solution does not always work for very small values of k, since we need a big enough n for the constant factors hidden in $O()$ notation above to not matter. For those values of k we can either use the solution above for the smaller constraints, or implement a very simple solution for very small constraints. For example, when $k \le \lfloor \frac{2025}{3} \rfloor^2$, we can just output a grid of separate 2×2 squares with empty rows and columns between them.

Also, since the resulting grid has $n+2$ columns, it means that we must have $n \le 2023$, so for the values of k that exceed the total number of rectangular walks in the 2023 times 2024 rectangle it will no longer be true that $k - x < 2n(n-1)$. However, one can more carefully examine how

much space does the greedy knapsack above need, and find that it can still fit inside the $n - O(\sqrt{n})$ untouched rows/columns at least for $k \leq 4.1948 \cdot 10^{12}$, which is enough to solve this problem.

Remarks

There are of course very many approaches that work in this problem; the above solutions are just one possibility.

The judges also had a solution that could get even closer to $f(2025)$, approximately to $k \leq \frac{1}{3}f(2024) + \frac{2}{3}f(2025)$. However, we felt that the $k \leq 4.194 \cdot 10^{12}$ constraint is similarly challenging, and did not want to restrict the possible approaches too much.

The fact that we are counting hollow rectangular walks instead of rectangles that also have only parks inside them did not actually affect the above solutions. We made them hollow so that counting rectangles takes $O(n^2 \log n)$ instead of $O(n^2)$, so hopefully simple hill-climbing solutions were less likely to work.

C Ads

Author: Andrea Ciprietti
Preparation: Tomaz Hocevar

The result is equal to $g - 1$ where g is the number of groups of videos. Therefore, you want to minimize the number of groups and you can do this by creating large groups: first of size 3, then of size 2 and of size 1 with whatever is left. To create groups of size 3, we will pick the shortest video (m) as the first element of the group, the longest video (M) as the last and for the middle one we choose the longest video of length x such that $m + x < k$. For groups of size 2, we pair the shortest video that is shorter than k with the longest video. We can simulate this greedy process with a tree data structure (e.g. multiset in C++) to solve the problem in $O(n \log n)$.

Let's prove that this greedy approach gives us an optimal solution.

- We can consider a version of the problem where we can choose to watch an ad early if we wish. The optimal solution will be equivalent to the original problem with forced ads. This is because we can move an ad (that we watched early) forward as much as possible and the number of groups in the following videos can't be larger by induction on the number of viewed ads.
- If the optimal solution contains a group of size 2 or 3, there is a solution that contains the shortest and longest video in the same group. If there wasn't, we can swap the longest video overall with the last video in the group of the shortest one (without affecting the number of groups).
- If it's possible to create a group of size 3, there exists an optimal solution with such group. Indeed, suppose there is an optimal solution without such group. Clearly all groups can't be of size 1 in an optimal solution so there must be at least one group of size 2. We know there is a solution with a group of size 2 containing the shortest and longest video. If it's possible to insert a third video, the solution can't degrade.
- If it's possible to construct a group of size 3, it can consist of the shortest (m) and longest (M) video in the first and last place, respectively. For the second video we can greedily choose the longest available video x such that $m + x < k$ to avoid triggering an early ad. We can prove that a greedy choice is valid with a similar swap argument as before.
- If it's not possible to construct groups of size 3, and it's possible to create a group of size 2, then there exists an optimal solution with at least a group of size 2. Otherwise, we would have only groups of size 1 and merging two directly decreases the number of groups.
- If it's not possible to construct groups of size 3, and it's possible to construct a group of size 2, then there exists an optimal solution where the shortest video is paired with the longest video. This can again be proved with a swap argument.

D Morse Code

Author: Jorke de Vlas
Preparation: Jorke de Vlas

Any assignment of Morse codes to characters can be represented with a binary tree, where going to a left child appends a dot and going to a right child appends a dash. We then associate each character with a leaf in the tree. This ensures that the codes will not be prefixes of each other.

Each vertex, both leafs and internal vertices, has a depth in the tree. For this depth to correspond to the length of the associated Morse code, we declare that going to a right child increases the depth by two. Once we know what an optimal binary tree looks like, we can greedily match the characters to the leafs by assigning the most frequent characters to the leafs with the smallest depth.

The key observation is that we initially do not need to know what exactly the tree looks like. We only need to know how many leafs and internal vertices exist at each depth. This allows us to use dynamic programming (DP) to determine an optimal tree bottom up. In each state, we keep track of:

- v_c, the number of vertices at the current depth level, where we can still decide whether they become leafs or internal vertices.
- v_n, the number of vertices at the next (deeper) depth level, where we can still decide whether they become leafs or internal vertices.
- a, the number of characters that have already been assigned a value

The value of this state is how much the expected transmission length increases due to dashes and dots that correspond to edges at the current level or deeper.

The base state is $DP[0,0,0] = 0$, since transmitting nothing costs nothing. The value we want to know is $DP[1,0,n]$, the cost to transmit all characters from depth zero where we only have a root vertex.

The transition function is a minimum of two values. To determine $DP[v_c, v_n, a]$, we can either make a vertex a leaf and assign a character to it, or we can determine that we do not want anymore leafs at this depth by splitting all remaining vertices into two new vertices and going to the next depth. In the first case, this results in the value $DP[v_c - 1, v_n, a - 1]$ with no additional costs. In the second case, this results in the value $DP[v_n + v_c, v_c, a]$ plus the sum of the frequencies of the least frequent a characters; the cost is caused by the fact that these characters are now placed one level deeper.

Finally, there are some boundary conditions where we assign the value infinity. If $v_c = v_n = 0$ and $a > 0$ the state is unsolvable since there are some unassigned characters left but no more leafs can be created. If $v_c + v_n > a$ then the state is trivially suboptimal since we will then obtain more leafs then needed.

After completing the DP, the final step is to actually construct an optimal tree. This can be done greedily by keeping track of two vectors of partial codes (one for the current depth, one for the next one) and backtracking through the DP.

Each parameter in the state description is bounded by n and the transition function is constant, so overall this results in an $O(n^3)$ algorithm.

E Porto Vs. Benfica

Author: Pedro Paredes
Preparation: Pedro Paredes

We start by making a few definitions that will help us:

Definition. Denote by $f(v)$ the minimum number of roads the supporters' club needs to travel if they start from vertex v and want to end up at vertex n, and the police can still block exactly one road.

So $f(1)$ is the answer to the problem and $f(n) = 0$.

Definition. Denote by $g(v, e)$ the shortest path from v to vertex n that doesn't use edge e and $g(v)$ as the maximum of $g(v, e)$ for all edges e adjacent to v.

In other words, $g(v)$ is the shortest path from v to n that doesn't use the edge that leads to the shortest path from v to n. Now we have the following:

Lemma. $f(v) = \max\{g(v), 1 + \min_{v \sim u} f(u)\}$, where $v \sim u$ denotes that the two vertices are adjacent.

Proof. It's easy to see that the police only block an edge when the supporters' club is on a vertex adjacent to that edge; otherwise, they could have waited until they were adjacent to that edge to block it. So, when the supporters' club is at some vertex v, the police have two choices (and they want to pick the one that maximizes the number of traversed roads): either block some edge adjacent to v, or not block any edge and let the supporters' club decide where to go. If they decide to block road e, then the supporters' club takes $g(v, e)$ roads to reach n, so the police might as well pick $g(v)$ to maximize this. If they decide not to block a road, the number of roads is 1 plus $f(u)$, where u is the vertex the supporters' club ends up at. Since they want to minimize the number of roads taken, they best pick u that minimizes $f(u)$.

Note that the values of g can easily be found in $O(m^2)$ time, by running a BFS per vertex v. This is too slow to solve this problem, so we will see later how to compute g more efficiently, which is the trickiest part. But first, let's see how to find efficiently the values of f assuming we have already computed g.

Computing f assuming we know g

To find f, we can use a greedy algorithm that works exactly the same way as Dijkstra's algorithm. Consider an array `f[]` that we will use to store the values of f. Set `f`$[n] = 0$ and `f`$[v] = g(v)$ for all other v. Now process each vertex in the following way. Pick the unprocessed vertex v with the lowest current value of `f`$[v]$. Process v by iterating through all vertices u adjacent to v and setting `f`$[v] = \max(g[v], \min(1 +$ `f`$[u]))$. Intuitively, this processing step amounts to applying the formula of f present in the lemma above, but considering only one pair of adjacent vertices at a time. To efficiently pick the unprocessed vertex v with the lowest current value of $f[v]$, we can use a min-heap ordered by `f`$[v]$, which needs to be updated accordingly.

This algorithm runs in $O(m \log n)$ time, since we process each vertex once and for each vertex we look at its neighbors and potentially update the elements in a heap containing at most n elements.

Lemma. After running the algorithm above, `f`$[v] = f(v)$.

Proof. The correctness of this algorithm follows from an argument very similar to the correctness of Dijkstra's algorithm. The key observation is that if v is an unprocessed vertex with the lowest

current value of $\texttt{f}[v]$, then at this point we have that $\texttt{f}[v] = \max(g(v), 1 + \min(f(u)))$, where the minimum is over all u that have been previously processed. Since v has the lowest current value of $\texttt{f}[v]$, none of the remaining unprocessed vertices could increase the value of $\texttt{f}[v]$, which means that the current value of $\texttt{f}[v]$ is exactly $f(v)$.

Computing g

Let's start with two quick definitions.

Definition. Denote by $\text{dist}(v, u)$ the distance between vertices v and u in the road network graph.

Definition. Let $b(v)$ be any neighbor of v such that $\text{dist}(v, n) = 1 + \text{dist}(b(v), n)$, and break ties arbitrarily.

So, we can think of $g(v)$ as the distance from v to n that doesn't use the edge $\{v, b(v)\}$.

Consider the BFS tree from n, which is the same as saying the tree consisting of the edges given by $\{v, b(v)\}$ for all $v \neq n$, and root the tree on n. Observe that the path that corresponds to $g(v)$ has to necessarily be a path that uses at least one non-tree edge (otherwise it would be a path that uses $\{v, b(v)\}$, which is invalid). Furthermore, such a path will look like the following: start at v, go down the subtree rooted on v some number of steps (potentially 0), take one non-tree edge that ends in a vertex that isn't in the subtree rooted at v and then take the shortest path from that vertex to n. Note that we can easily precompute the shortest path from any vertex to n by running a single BFS from n.

To prove the above, note that once we are outside of the subtree rooted at v we are free to take the shortest path to n since it won't use the $\{v, b(v)\}$ edge. Additionally, note that we don't want to ever take a non-tree edge to end up at another vertex in the subtree of v, since we can always take the tree path and that will always be at most as short (by definition of BFS tree).

So we are left with computing for each v the shortest among all paths that take some number of steps down the tree, then takes some non-tree edge, and then takes the shortest path to n. Let's first see an inefficient way of doing to, and then we'll make it efficient.

For each vertex v, compute a list of all the paths of the desired form, and store this in a set (so each vertex gets a set). In particular, we store pairs of two things in this set: the distance to n of the corresponding path, the endpoint of the non-tree edge we are taking. We can do so recursively (in a DFS fashion), so let's start by considering the leaves of the tree.

If v is a leaf, then the path can't go down the tree, so it is of the form "take some non-tree edge and then take the shortest path to n". For each non-tree edge $\{v, u\}$, add the pair $(1 + \text{dist}(u), u)$ to the set. So now we know that $g(v)$ is the minimum element in the set.

If v isn't a leaf, first recursively compute the distances of all of the children of v. Initialize the set corresponding to v by going through each non-tree edge $\{v, u\}$, and adding the pair $(1+\text{dist}(u), u)$. Now "merge" this set with all the sets of all of v's children. To merge two sets, we take all pairs (d, u) of the children sets and add $(d + 1, u)$ to v's set, which corresponds to extending each path from each of the children by taking the edge from v to them. Suppose the minimum element is (d, u). Then u could be in the subtree rooted at v, which is an invalid path. If that's the case, we delete (d, u) from the set and keep deleting the top element until we find one which isn't in the subtree of v. Note that this doesn't affect the computation of ancestors of v since u would be in their subtree too. To determine whether u is in the subtree of v we can use DSU (Disjoint Set Union). Initially, all vertices are their own set, and as we go down the tree we merge v with its children. Like before, now we know that $g(v)$ is the minimum element in the remaining set.

We are almost done, but there is one inefficient step here: merging the sets of the children of v could take $O(n)$ time since each set could have up to n elements. To implement this step efficiently, we can do "small-to-large merging", just like the union by size merge in a DSU data structure. When merging two sets, don't touch the largest of the two, and copy the elements of the smallest one into the largest one. However, recall that when merging two sets, we take all pairs (d, u) of the children sets and add $(d + 1, u)$ to v's set, so we need to potentially alter the elements in the largest set if this is one of the children's sets. We can do this by assuming that each set comes with a "modifier", which is an integer m such that if we have an element (d, u) in the set, the real distance is $d + m$. Intuitively, this modifier acts as a "lazy propagator", so that we don't have to actually change the elements in the children's sets. When we take a set from a child, we first increment its modifier, and then merge its set with v's set.

The total running time of this step is $O(n \log^2 m)$, since the small-to-large merge makes sure we only move an element between sets $O(\log m)$ times, and each move costs $O(\log m)$.

F Mascot Naming

Author: Federico Glaudo
Preparation: Federico Glaudo

This problem requires constructing a string that contains each of the given strings $s_1, s_2, \ldots, s_n$ as subsequences but does not contain t as a subsequence.

Key Observation. If t is a subsequence of any s_i, then any string containing s_i as a subsequence must also contain t. In this case, the answer is `NO`. Otherwise, we can construct a valid string using a greedy approach.

Checking for Subsequences. Before solving the main problem, we describe a simple method to check if a string b is a subsequence of another string a. We iterate through a, trying to match its characters to b in order:

- If the first characters of a and b match, remove the first character from both.
- Otherwise, remove only the first character from a.
- Repeat until a is empty.
- If b becomes empty, it was a subsequence of a; otherwise, it was not.

Constructing the Answer. We build the answer string ans by appending characters one by one. Initially, ans is empty. We repeat the following steps until all s_i are empty:

- If there exists a character c such that at least one s_i starts with c and t does not start with c, append c to ans and remove the first character from all s_i that start with c.
- Otherwise, all s_i start with the same character as t. Append this character to ans and remove the first character from all s_i and t.

At the end of this process, ans contains all s_i as subsequences.

Let us understand why t is not a subsequence of ans. Let s_i be the last string to become empty during the process. Consider the steps taken for s_i and t alone. The sequence of character deletions mirrors the subsequence checking algorithm. Since t is not a subsequence of s_i, it remains non-empty at the end, ensuring that t is not a subsequence of ans either.

Thus, if t is not a subsequence of any s_i, the algorithm constructs a valid ans, and the answer is `YES`.

The solution described is linear in the total length of all strings given in input.

G A Very Long Hike

Author: Federico Glaudo
Preparation: Federico Glaudo

Before diving into the solution, let us make a few key observations to build intuition for tackling the problem.

- Our goal is to count the number of distinct cells that can be reached within a time limit of 10^{20}. To gain insight into this, consider first the simpler question: given a distant cell (x, y) (where x, y are large), how can we estimate the shortest time required to reach it from the origin?
- Since we do not need an exact answer but only a sufficiently precise approximation, we can reframe the previous question: how can we efficiently approximate the distance from the origin to (x, y)?
- Consider any path from $(0, 0)$ to (x, y). If this path visits the same position modulo n at both the i-th and j-th steps, then removing the portion of the path between these two steps does not change the total cost of the remaining path.

The last observation is particularly powerful and will be crucial in approximating distances efficiently. We will leverage it to resolve the subproblem identified in the first two observations.

Afterward, we will use this approximation to count (approximately) the number of reachable cells within the given time constraint.

Finally, we will describe how to translate these results into an algorithm with time complexity $O(n^5)$.

It is worth noting that this editorial is relatively long because we rigorously prove that our approximations maintain a relative error smaller than 10^{-6}. However, in practice, we expect that any reasonable implementation with sufficient numerical precision will be far more accurate than required. As such, contestants do not need to be overly concerned about precision.

Lastly, many of the statements we prove here are significantly easier to intuit than to rigorously justify. We recommend that readers focus on understanding the core ideas on their first read and not get too caught up in the technical details. If you grasp the reasoning behind Lemma 5 and Lemma 6, you should already have the essential insights needed to solve the problem.

Approximating the distance

Let us denote by $D := 1545+1$ the maximum distance between two adjacent cells (i.e., the maximum altitude difference plus one).

For any integers i, j with $|i| + |j| \leq n$, let t be the minimum number such that, for some (a, b), the distance from (a, b) to $(a + in, b + jn)$ is t. Define $T(i, j) := t/n$. This captures the idea that moving by (i, j) costs $T(i, j)$.

For any x, y, we define $f(x, y)$ as the infimum of $\sum_{|i|+|j|\leq n} \lambda_{ij} T(i, j)$, for any nonnegative real numbers $\lambda_{ij} \geq 0$, under the constraint that $\sum_{|i|+|j|\leq n} \lambda_{ij}(i, j) = (x, y)$. Observe that $|x| + |y| \leq f(x, y) \leq D(|x| + |y|)$.

Lemma 1. For any cell (x, y) such that both coordinates are divisible by n, we have

$$\text{dist}((0,0),(x,y)) \geq f(x,y).$$

Proof. We show it by induction on the distance from $(0,0)$ to (x,y).

Consider a shortest path from $(0,0)$ to (x,y). Since $(0,0) \equiv (x,y) \pmod{n}$, we can consider the two closest cells (x_1,y_1) and (x_2,y_2) along the path such that $(x_1,y_1) \equiv (x_2,y_2) \pmod{n}$. Let $(x',y') = (x_2 - x_1, y_2 - y_1)$. By construction, the segment of the path from (x_1,y_1) to (x_2,y_2) contains no two positions (besides the endpoints) that are congruent modulo n, so it must have at most n^2 steps. Thus, we obtain $|x'| + |y'| \leq n^2$.

Using our third key observation from the introduction, we deduce that

$$\text{dist}((0,0),(x,y)) = \text{dist}((x_1,y_1),(x_2,y_2)) + \text{dist}((0,0),(x-x',y-y')).$$

By the definition of T, we obtain

$$\text{dist}((x_1,y_1),(x_2,y_2)) \geq nT(x'/n, y'/n).$$

By the induction hypothesis, we also have

$$\text{dist}((0,0),(x-x',y-y')) \geq f(x-x',y-y').$$

Combining these inequalities, we conclude

$$\text{dist}((0,0),(x,y)) \geq nT(x'/n,y'/n) + f(x-x',y-y') \geq f(x,y).$$

Lemma 2. For any cell (x,y), we have

$$\text{dist}((0,0),(x,y)) \geq f(x,y) - 2Dn.$$

Proof. Let $(\bar{x},\bar{y})$ be a cell such that its coordinates are divisible by n and the Manhattan distance from (x,y) to $(\bar{x},\bar{y})$ is $\leq n$. We have

$$\text{dist}((0,0),(x,y)) \geq \text{dist}((0,0),(\bar{x},\bar{y})) - \text{dist}((\bar{x},\bar{y}),(x,y)) \geq f(\bar{x},\bar{y}) - Dn.$$

Furthermore, observe that

$$f(x,y) \leq f(\bar{x},\bar{y}) + f(x-\bar{x},y-\bar{y}) \leq f(\bar{x},\bar{y}) + T(x-\bar{x},y-\bar{y}) \leq f(\bar{x},\bar{y}) + Dn.$$

Joining the last two inequalities, we infer,

$$\text{dist}((0,0),(x,y)) \geq f(x,y) - 2Dn.$$

We have shown that f provides a lower bound for the distance of (x,y) to the origin. Now we turn to proving that it provides an upper bound.

Lemma 3. Fix a nonnegative integer $k \geq 0$ and two integers i, j with $|i| + |j| \leq n$. For any cell (x,y), we have

$$\text{dist}((x,y),(x,y)+k(in,jn)) \leq knT(i,j) + 2Dn.$$

Proof. Without loss of generality, we may assume that $(x,y) = (0,0)$.

By definition of $T(i,j)$, there exists (a,b) such that $\text{dist}((a,b),(a,b)+(in,jn)) = T(i,j)$. Without loss of generality, we may assume that $|a|+|b| \le n$.

Consider the path that goes from $(0,0)$ to (a,b), then to $(a,b)+k(in,jn)$ repeating k times the path that defines $T(i,j)$, and finally from $(a,b)+k(in,jn)$ to $k(in,jn)$. The total length of this path is at most $Dn + knT(i,j) + Dn$ as desired.

Now that we have seen in which cases T provides an upper bound, we deduce how f always provides an upper bound. One can show (it follows from the argument in the next section of this editorial) that the value of f does not change if we require that at most two values of λ_{ij} are nonzero.

Lemma 4. For any cell (x,y), we have

$$\text{dist}((0,0),(x,y)) \le f(x,y) + 2Dn(n+1).$$

Proof. Let $(x,y) = a(i,j) + a'(i',j')$ so that $a, a' \ge 0$ and $f(x,y) = aT(i,j) + a'T(i',j')$. Consider the cell

$$(\bar{x},\bar{y}) := \lfloor a/n \rfloor (in,jn) + \lfloor a'/n \rfloor (i'n, j'n).$$

Thanks to the previous lemma, we have

$$\text{dist}((0,0),(\bar{x},\bar{y})) \le \lfloor a/n \rfloor nT(i,j) + \lfloor a'/n \rfloor nT(i',j') \le f(x,y) + 2Dn.$$

Since the Manhattan distance from $(\bar{x},\bar{y})$ to (x,y) is at most $2n^2$, we deduce that

$$\text{dist}((0,0),(x,y)) \le \text{dist}((0,0),(\bar{x},\bar{y})) + \text{dist}((\bar{x},\bar{y}),(x,y)) \le f(x,y) + 2Dn + 2n^2 D.$$

The results of this section can be summarized in the following statement.

Lemma 5. For any cell (x,y), we have

$$f(x,y) - 2Dn \le \text{dist}((0,0),(x,y)) \le f(x,y) + 2Dn(n+1).$$

Counting the cells with distance $\le R$

Let $R = 10^{20}$. Our goal is to count the number of cells (x,y) with $\text{dist}((0,0),(x,y)) \le R$. Let $q(r)$ be the number of cells (x,y) with $f(x,y) \le r$. In view of **Lemma 5**, we have (setting $R_1 := R - 2Dn(n+1)$ and $R_2 := R + 2Dn$),

$$q(R_1) \le \#\{(x,y) : \text{dist}((0,0),(x,y)) \le R\} \le q(R_2). \quad (*)$$

Our plan is to first compute $q(r)$, and then to show that the relative discrepancy between $q(R_1)$ and $q(R_2)$ is small enough. In order to fully understand the behavior of $q(r)$ we must investigate better how to compute the function $f(x,y)$. We are going to make some algebraic manipulations that transform the computation of f into finding whether a point belongs to a convex polygon.

Recall that $f(x,y)$ is defined as

$$\inf \left\{ \sum_{|i|+|j| \le n} \lambda_{ij} T(i,j) : \lambda_{ij} \ge 0 \text{ and } \sum_{|i|+|j| \le n} \lambda_{ij}(i,j) = (x,y) \right\}.$$

Observe that this formula is equivalent to (for $(x,y) \neq 0$)

$$\inf\left\{\sum_{|i|+|j|\leq n} \mu_{ij} : \mu_{ij} \geq 0 \text{ and } \sum_{|i|+|j|\leq n} \mu_{ij}\left(\frac{i}{T(i,j)}, \frac{j}{T(i,j)}\right) = (x,y)\right\}.$$

Define $p_{ij} := \left(\frac{i}{T(i,j)}, \frac{j}{T(i,j)}\right)$. The latter formula is equivalent to

$$\inf\left\{t \geq 0 : \text{ there are } \mu_{ij} \geq 0 \text{ with } \sum_{|i|+|j|\leq n} \mu_{ij} \leq 1 \text{ such that } \sum_{|i|+|j|\leq n} \mu_{ij} p_{ij} = \frac{1}{t}(x,y)\right\}.$$

Let P be the convex hull of the points p_{ij}. We have proven that $f(x,y)$ is the minimum value $t \geq 0$ such that $(x/t, y/t)$ belongs to P. Therefore, we have proven that

Lemma 6. Let P be the convex hull of the points $p_{ij} := \left(\frac{i}{T(i,j)}, \frac{j}{T(i,j)}\right)$. For any $r > 0$, the number of lattice points in rP (here, rP represents the scaling of P by a factor r) coincides with $q(r)$.

So, it remains only to (approximately) compute the lattice points inside rP. The standard way to approximate the number of lattice points in a convex subset is by computing the area of the subset. Let us check that it is precise enough in our setting. For any convex polygon, we have

Lemma 7. Let P be an arbitrary convex polygon. We have

$$\text{area}(P) - \text{perimeter}(P) - \pi \leq \#\{(x,y) \in P : x, y \text{ are integers}\} \leq \text{area}(P) + \text{perimeter}(P) + \pi.$$

Proof. Denote by $d(x,X)$ the Euclidean distance between a point x and a subset X of the plane. For $r > 0$, let P_r be the *enlargement* of P given by $\{p : d(p,P) \leq r\}$. For $r < 0$, let P_r be the *reduction* of P given by $\{p : d(p,P^c) > r\}$. Observe that if r_1, r_2 have the same sign, then $(P_{r_1})_{r_2} = P_{r_1+r_2}$.

Consider the disjoint squares centered at all lattice points in P. We have

$$P_{-\frac{1}{\sqrt{2}}} \subseteq [-0.5, 0.5) \times [-0.5, 0.5) + \{(x,y) \in P : x, y \text{ integers}\} \subseteq P_{\frac{1}{\sqrt{2}}}.$$

By taking the area of these three domains, we obtain

$$\text{area}\left(P_{-\frac{1}{\sqrt{2}}}\right) \leq \#\{(x,y) \in P : x, y \text{ integers}\} \leq \text{area}\left(P_{\frac{1}{\sqrt{2}}}\right).$$

Thus, we have reduced the problem to estimating the area of P_r for r positive and negative. The estimates necessary to conclude the current lemma are proven in the next lemma. We don't need the lower bound on the perimeter and the upper bound on the area for $r < 0$ that are stated in the next lemma; we decided to include them for completeness.

Lemma 8. For $r > 0$, we have

$$\text{perimeter}(P_r) = \text{perimeter}(P) + 2\pi r, \quad \text{area}(P_r) = \text{area}(P) + r \cdot \text{perimeter}(P) + \pi r^2,$$

while for $r < 0$, we have

$$2\sqrt{\pi}\sqrt{\text{area}(P_r)} \leq \text{perimeter}(P_r) \leq \text{perimeter}(P), \quad \text{area}(P) + r \cdot \text{perimeter}(P) - \pi r^2 \leq \text{area}(P_r) \leq \pi(R+r)^2,$$

where $R > 0$ is such that $\text{area}(P) = \pi R^2$.

Proof. Let us observe that, for any r, $\frac{d}{dr}\text{area}(P_r) = \text{perimeter}(P_r)$. The formulas for the area follow from those for the perimeter using this differential equation. The formula for the perimeter when $r > 0$ is left to the reader. For $r < 0$, since $P_r \subseteq P$ we deduce $\text{perimeter}(P_r) \leq \text{perimeter}(P)$ and the lower bound for $\text{perimeter}(P_r)$ is the isoperimetric inequality.

Concatenating Lemma 6 and Lemma 7 together with $(*)$, we have

$$R_1^2\text{area}(P) - R_1\text{perimeter}(P) - \pi \leq \#\{(x,y) : \text{dist}((0,0),(x,y)) \leq R\} \leq R_2^2\text{area}(P) + R_2\text{perimeter}(P) + \pi.$$

If the right-hand side over the left-hand side is smaller than $1 + 10^{-6}$, a correct answer to the problem is given by $R^2\text{area}(P)$. In order to show it, we shall prove that the perimeter of P is not disproportionately larger than the area of P. (One could prove a stronger lemma, showing that the ratio between perimeter and area is $O(D)$ instead of $O(D^2)$, but it is harder to show and we don't need it.)

Lemma 9. We have $\text{perimeter}(P) \leq 4D^2\text{area}(P)$ and also $\text{area}(P) \geq 2D^{-2}$.

Proof. By definition of P, it is not hard to check that

$$\{(x,y) : |x| + |y| \leq D^{-1}\} \subseteq P \subseteq \{(x,y) : |x| + |y| \leq 1\}.$$

In particular, we deduce that $\text{perimeter}(P) \leq 4\sqrt{2} \leq 8$ and $\text{area}(P) \geq 2D^{-2}$.

Thanks to this lemma, we obtain

$$(R_1^2 - 4D^2R_1 - 2D^2)\text{area}(P) \leq \#\{(x,y) : \text{dist}((0,0),(x,y)) \leq R\} \leq (R_2^2 + 4D^2R_2 + 2D^2)\text{area}(P).$$

Given the constraints of the problem, one can check that

$$\frac{R_2^2 + 4D^2R_2 + 2D^2}{R_1^2 - 4D^2R_1 - 2D^2} < 1 + 10^{-12},$$

so a correct answer to this problem is given by $R^2 \cdot \text{area}(P)$.

Computing the polygon P fast enough

Solving the problem is straightforward once we have computed $T(i,j)$ for the $O(n^2)$ pairs (i,j) such that $|i| + |j| \leq n$. Indeed, once we have these values:

- We can compute the family of points p_{ij} in $O(n^2)$.
- We can compute its convex hull in $O(n^2 \log(n))$.
- Finally, computing the area of the resulting polygon is straightforward.

To compute $T(i,j)$ for all i,j with $|i| + |j| \leq n$, a naive approach would be:

- Iterate over all $O(n^2)$ interesting pairs (i,j).
- For each such pair, iterate over all $O(n^2)$ interesting pairs (a,b).
- Compute the distance between (a,b) and $(a+in, b+jn)$ using Dijkstra's algorithm, which takes $O(n^4 \log(n))$.

This results in a total complexity of $O(n^8 \log(n))$, which is too slow.

Optimization 1: Batch processing in Dijkstra. Instead of computing $T(i, j)$ separately for each pair, we observe that once (a, b) is fixed, we can compute the answer for all relevant (i, j) in one run of Dijkstra's algorithm. This reduces the complexity to $O(n^6 \log(n))$, which is still likely too slow. Let us remark that, when running Dijkstra's algorithm, we can completely ignore all cells at a Manhattan distance greater than n^2 from (a, b). This is because we are only interested in paths where no two points share the same position modulo n. Otherwise, the path could be decomposed into multiple shorter segments, each obeying this property.

Optimization 2: Reducing the search space using modulo n properties. A key observation is that any path whose endpoints are equivalent modulo n must contain a cell where at least one coordinate is divisible by n. Thus, we may assume that either a or b is 0. This reduces the complexity to $O(n^5 \log(n))$, which might be sufficient to get accepted.

Optimization 3: Removing the logarithmic factor. To further improve performance, note that the maximum weight is D. We can replace Dijkstra's priority queue with a bucket-based shortest path algorithm (similar to 0-1 BFS). This reduces the complexity per Dijkstra run to $O(n^5 + D)$.

H Statues

Author: Federico Glaudo, Giovanni Paolini
Preparation: Giovanni Paolini

Let $d_0 = a + b$ be the distance between the first and the n-th statue. We are going to prove that a valid arrangement exists if and only if the following two conditions are satisfied:

1. $d_0 + d_1 + \cdots + d_{n-1}$ is even;
2. $d_i \le d_0 + \cdots + d_{i-1} + d_{i+1} + \cdots + d_{n-1}$ for all $0 \le i \le n-1$.

To start, let us prove that the two conditions above are necessary. Suppose there is a valid arrangement $s_1 = (x_1, y_1)$, $s_2 = (x_2, y_2)$, …, $s_n = (x_n, y_n)$. Then

$$d_0+d_1+\cdots+d_{n-1} = (|x_n - x_1| + |y_n - y_1|)+(|x_1 - x_2| + |y_1 - y_2|)+\cdots+(|x_{n-1} - x_n| + |y_{n-1} - y_n|)\,;$$

when considered mod 2, the absolute values can be omitted and all terms cancel out, so the result is even (condition 1). For every $0 \le i \le n-1$, the triangle inequality ensures that the distance d_i between s_i and s_{i+1} (where s_0 is considered to be the same as s_n) is not greater than the sum of all other distances (condition 2).

Conversely, we now show by induction on n that conditions 1 and 2 are sufficient to construct a valid arrangement. Our proof will effectively yield an $O(n)$ algorithm to construct a valid arrangement.

If $n = 2$, condition 2 tells us that $d_0 \le d_1$ and $d_1 \le d_0$, so $d_0 = d_1$. Then the arrangement $s_0 = (0, 0), s_1 = (a, b)$ is valid.

Suppose now that $n \ge 3$. Our aim is to determine a placement $s_{n-1} = (a', b')$ for the $(n-1)$-th statue such that:

- distance$(s_{n-1},\, s_n) = d_{n-1}$;
- if we define $\hat{d}_0 = $ distance$(s_1,\, s_{n-1})$, then the distances $\hat{d}_0, d_1, \ldots, d_{n-2}$ satisfy conditions 1 and 2.

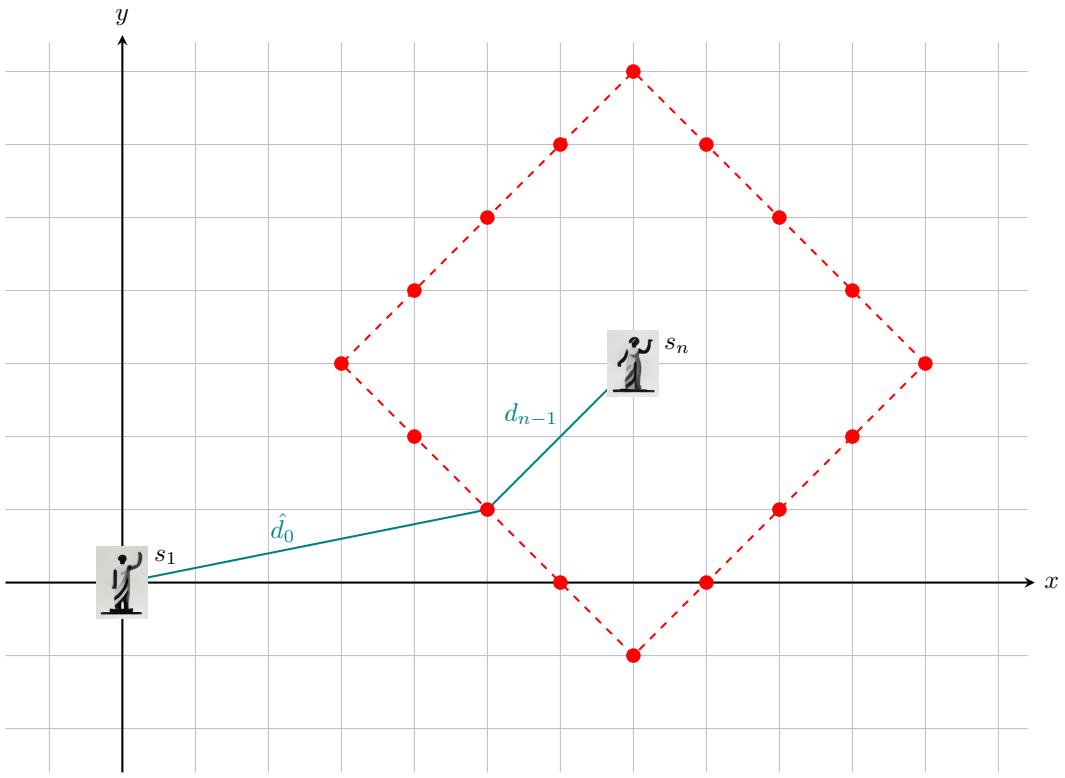
If we manage to achieve this, then we can inductively arrange the statues $1, \ldots, n-1$. (Note that the coordinates a' or b' can be negative, but this is not a problem when applying the induction hypothesis, as our solution does not require a and b to be nonnegative.)

For any choice of s_{n-1}, note that $d_0 + \hat{d}_0 + d_{n-1}$ is necessarily even, because $d_0, \hat{d}_0, d_{n-1}$ are the distances between the three points s_1, s_{n-1}, s_n. Therefore,

$$\hat{d}_0 + d_1 + \cdots + d_{n-2} \equiv d_0 + d_1 + \cdots + d_{n-1} \equiv 0 \pmod 2,$$

so condition 1 is satisfied by $\hat{d}_0, d_1, \ldots, d_{n-2}$. The harder part is to ensure condition 2.

As s_{n-1} varies among all integer points with distance$(s_{n-1},\, s_n) = d_{n-1}$, the distance $\hat{d}_0$ takes all values between $|d_0 - d_{n-1}|$ and $d_0 + d_{n-1}$ having the same parity as $d_0 + d_{n-1}$. The possible locations of s_{n-1} are shown in red in the picture below.



We are going to show that the value $\hat{d}_0 = \min(d_0 + d_{n-1},\, d_1 + \cdots + d_{n-2})$ belongs to the admissible range $\{|d_0 - d_{n-1}|, \ldots, d_0 + d_{n-1}\}$, has the correct parity, and makes the sequence $\hat{d}_0, d_1, \ldots, d_{n-2}$ satisfy condition 2.

Case 1: $\hat{d}_0 = d_0 + d_{n-1} \le d_1 + \cdots + d_{n-2}$. Obviously, $d_0 + d_{n-1}$ belongs to the admissible range and has the correct parity. The sequence $\hat{d}_0, d_1, \ldots, d_{n-2}$ satisfies condition 2 for $i = 0$ because $\hat{d}_0 \le d_1 + \cdots + d_{n-2}$. For $i \ge 1$, we need to check that

$$d_i \le \hat{d}_0 + d_1 + \cdots + d_{i-1} + d_{i+1} + \cdots + d_{n-2} = (d_0 + d_{n-1}) + d_1 + \cdots + d_{i-1} + d_{i+1} + \cdots + d_{n-2},$$

and this is ensured by condition 2 for the original sequence $d_0, \ldots, d_{n-1}$.

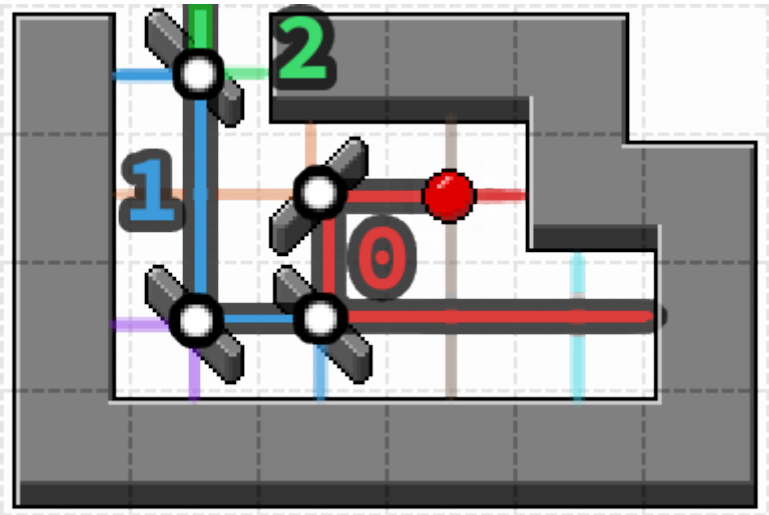
Case 2: $\hat{d}_0 = d_1 + \cdots + d_{n-2} \le d_0 + d_{n-1}$. For $\hat{d}_0$ to belong to the admissible range, we need to check that $d_1 + \cdots + d_{n-2} \ge |d_0 - d_{n-1}|$. This is ensured by condition 2 for the original sequence $d_0, \ldots, d_{n-1}$ for $i = 0$ and $i = n-1$. Condition 1 for the original sequence ensures that $d_1 + \cdots + d_{n-2}$ has the same parity as $d_0 + d_{n-1}$.

We now check condition 2 for the sequence $\hat{d}_0, d_1, \ldots, d_{n-2}$. For $i = 0$, we are done because $\hat{d}_0 = d_1 + \cdots + d_{n-2}$. For $i \ge 1$, condition 2 is satisfied because $d_i \le \hat{d}_0$.

I Pinball

Author: Lucian Bicsi
Preparation: Lucian Bicsi

Let's decompose the whole grid into regions delimited by the movement of the ball without altering the walls. These regions are depicted in the picture below by colored lines. The path described by the solution is bolded.



Let's build a graph over the set of all regions where we add an edge (i, j) if regions i and j share an oblique wall. In the picture above, the **red** and **blue** regions will be connected by an edge. Similarly, the **blue** and **purple** regions, as well as the **blue** and **green** regions and the **red** and **orange** regions are adjacent in this formulation.

Furthermore, let's compute the minimum distance between any of the two initial regions where the ball is located, and any of the regions that go outside the grid in this above described graph. Intuitively, this distance is a lower bound for our answer, as destroying k walls can only let the ball travel to regions at distance at most k from the initial regions. For a more formal proof, consider the relaxation of the problem where instead of destroying k walls, we can choose k walls and allow each of them to independently be *active* or *inactive* at any moment in time. Clearly, under this relaxation, the ball can only reach regions at distance at most k from its initial position.

Next, we can prove that this lower bound is, in fact, the answer to our problem, by providing a construction that allows the ball to escape under this minimum number of destroyed walls. There are multiple ways of solving the reconstruction problem. The easiest is perhaps to first find any sequence of k regions that end up with the ball escaping. Then, starting from the most distant (k-th) region, simulate the game in reverse order, and once you first reach the oblique wall that connects the k-th and $(k-1)$-th regions, "destroy" such wall (in reverse, the correct formulation is to "restore" the already-destroyed wall), and continue the process until the ball reaches its initial position.

Finally, an important aspect to quickly solve this problem is implementation. In order to simplify the implementation, one may skip constructing the graph described above and instead use it implicitly

by keeping information over triplets of the form (i, j, d) where (i, j) is the position of the ball, and $d \in \{\texttt{U}, \texttt{D}, \texttt{L}, \texttt{R}\}$ is the direction where the ball is heading, and using 0-1 BFS or even Dijkstra to compute the minimum number of walls that need to be destroyed to reach any such state.

J The Ultimate Wine Tasting Event

Author: Andrea Ciprietti
Preparation: Andrea Ciprietti

We will find a simple necessary and sufficient condition on the string s for the operation to be successful. Then, it will be trivial to check whether this condition is satisfied.

First, suppose that it is possible to rearrange the bottles as described in the statement. That is, there esist two subsets $A = \{a_1, \ldots, a_n\}$ and $B = \{b_1, \ldots, b_n\}$ of $\{1, 2, \ldots, 2n\}$, each of size n and disjoint, such that swapping s_{a_i} and s_{b_i} (assuming the elements are sorted in each subset) results in the string `WW`$\ldots$`WWRR`$\ldots$`RR`.

Let h be the last index such that $a_h \leq n$, and similarly let k be the last index such that $b_k \leq n$. If either of them do not exist, set it equal to 0. Note that, by definition, $h + k = n$. Modulo swapping A and B, we can assume that $h \leq k$. Then, upon swapping a_i with b_i for $1 \leq i \leq n$, all the bottles at indices $a_1, \ldots, a_h$ will occupy positions among the leftmost n, which means that $s_{a_i} =$ `W` for all $1 \leq i \leq h$. Also, after swapping, all bottles at indices $b_1, \ldots, b_h$ will occupy positions among the first n, and therefore $s_{b_i} =$ `W` for $1 \leq i \leq h$. On the other hand, the bottles at indices $b_{h+1}, \ldots, b_k$ will occupy positions greater than n, so that $s_{b_i} =$ `R` for each $h + 1 \leq i \leq k$. This implies that, among the first n bottles, the white ones are $2h$ and the red ones are $k - h = n - 2h$.

Finally, consider the first h bottles in the initial arrangement. Each of these belongs to either $\{a_1, \ldots, a_h\}$ or $\{b_1, \ldots, b_h\}$, and therefore, when swapped, it will end up in one of the first n positions. This implies that $s_i =$ `W` for $1 \leq i \leq h$.

We deduced that a necessary condition is: the number of white bottles in the first half is even, say $2h$, and the first h bottles are white. Note that the number of red bottles in the second half is also $2h$ and, by the same argument, the last h bottles have to be red.

Let's now prove that this condition is also sufficient. We construct A and B as follows:

- A contains the first h positions (which are white wines), and the first $n - h$ positions with red wines.
- B contains everything else, that is, all positions with white wines except the first h, and the last h positions (with red wines).

It is easy to see that the operation of swapping A and B places all white wines in the first half and all red wines in the second half.

Checking whether this condition is satisfied is trivial and can be done in time $O(n)$.

Remark. Arrangements such as `WWRWWRRRRW` ($n = 5$), where only the "first-half" condition is satisfied, do not work. One needs to check that both the white wines in the first half as well as the red wines in the second half satisfy the condition.

K Amusement Park Rides

Author: Egor Kulikov
Preparation: Egor Kulikov

Assume that among the numbers a_i there are k distinct values, denoted by $p_1, p_2, \ldots, p_k$. Suppose each p_j appears q_j times in the sequence $\{a_i\}$.

We now construct a flow graph as follows:

1. Define the vertex set as $V = \{S, T\} \cup \{A_1, A_2, \ldots, A_k\}$.
2. For each vertex A_i (with $1 \le i \le k$), add an edge from S to A_i with capacity q_i.
3. For each positive integer i, introduce a vertex B_i. For every vertex A_j (with $1 \le j \le k$), if $i \equiv 0 \pmod{p_j}$, then add an edge from A_j to B_i with capacity 1. In addition, add an edge from B_i to T with capacity 1.

We continue adding vertices B_i (in increasing order of i) until the maximum flow from S to T reaches n. The answer to the problem is the largest index i for which a vertex B_i was added.

This approach has a running time of $O(\text{ans} \cdot E)$, where ans denotes the final value of i and E is the number of edges. However, this is too slow since it requires creating a vertex B_i for every positive integer i.

We can improve efficiency by avoiding the creation of every B_i. Instead, we maintain a mapping $M : \mathbb{Z}^+ \to \{\text{indices}\}$, where $M(i)$ is a set of indices. Initially, for each $1 \le j \le k$, we set $M(p_j) = \{j\}$.

At each step, perform the following:

- Find the smallest i for which $M(i)$ is nonempty.
- Add the vertex B_i along with its edge to T and add edges from A_j to B_i for every $j \in M(i)$.
- If adding these edges increases the maximum flow, then for each $j \in M(i)$ insert j into $M(i+p_j)$. This means that a future vertex B_{i+p_j} may also need to be connected to A_j. Otherwise, if the maximum flow does not increase, no additional edges from A_j (for $j \in M(i)$) will be needed.
- Finally, clear the set $M(i)$.

Note that we examine at most $n+k$ distinct values of i (because in each step we either increase the maximum flow or remove at least one element from $\bigcup_i M(i)$). Therefore, the overall time complexity of the solution is $O(n \cdot E)$, and it can be shown that $E = O(n \log n)$.