A Light Inconvenience (light)

by Fabian Gundlach

Subtask 1. Only one call to *leave* per testcase

For this subtask, it will be convenient to reindex the performers, numbering them from *right to left*, starting at 1. (Any call to *join* or *leave* will then result in an index shift). Moreover, we will always only remember the sequence $1 = f_1 < f_2 < \cdots$ of performers whose torches are on fire.

Let's think about this problem greedily: what condition do we need to put on the f_i's to be able to survive the next act?

- We can *always* handle the next call to *join* with $t_a = p_a$: we can simply shift everything to the right by p_a (i.e. keeping the previous f_i's), adding further fires on the left if desired.
- We can handle a single call to *leave* by announcing $t_a = p_a$ if and only if $f_i \le 2f_{i-1} + 1$ *and* the leftmost torch is on fire: the first condition is necessary since having all performers starting at f_i leave will have $p_a = f_i$, and we have to light the torch of performer $f_i + 1$; the second condition is necessary to handle all but the leftmost performer leaving.

  On the other hand, it is not hard to check that this also sufficient to handle any call to *leave*: if we just announce p_a, this is enough to light the torch of the rightmost performer remaining on stage.

This suggests taking $f_i = 2^{i-1}$ (plus the leftmost performer) before the first call to *leave*, and then e.g. extinguishing all torches except the rightmost one. For any call to *join* after that, we can simply shift everything to the right again without adding further flaming torches. This will never have more than $\log_2 N+1$ torches on fire at the same time, which easily fits into the limit specified in the statement.

Subtask 2. $N \le 700$

Let us index the torches as in the task statement again. It suffices to announce $f_1 = 1, f_2 = 6, f_2 = 11, \dots, f_k = n$ (where n is the number of performers currently in line) at the end of any act; this will in fact only require $t_a = 4$ for any *leave* or $t_a = 4p_a$ for *join*. The maximum number of flaming torches is $\frac{1}{5}N + 1$.

Subtask 3. $N \le 5\,000$

The previous solution does not make use of the fact that we are allowed to announce large numbers whenever p_a is sufficiently large, giving us much more freedom.

Intuitively, we should exploit this by having much more fires near the right end of the line. Consider the following straegy: fix a number K and split the performers into blocks of length K, starting on the left. We will always guarantee that the leftmost performer in each block (i.e. performers $1, K + 1, 2K + 1, \dots$) holds a flaming torch. Moreover, every torch in the last block should be on fire.

Again any call to *join* would be trivial to handle. If we have a call to *leave*, however, we would run into a problem to uphold our invariant when all $0 < k \le K$ performers of the last block leave, as we will need $t_a = K - 1$ to fill up the last block. To avoid this issue, we will always hold some additional torches in the penultimate block: namely, we will have the leftmost ℓ torches on fire, where ℓ is the number of torches missing in the last block. As a result:

- If $j < k$ performers leave, we just announce $t_a = j$ to add fires to the penultimate block.
- If $k \le j < K$ performers leave, announcing $t_a = j$ is now enough to fill up the new final block (and it is never a problem to add the fires on the left of the new penultimate block).

- If at least K performers leave, we are free to do whatever we want in any block anyhow.

Again, the invariant is also trivial to uphold in *join*.

The maximum number of torches on fires is $N/K + K$, so taking $K = 75$ suffices to solve this subtask.

Subtask 4. $N \le 25\,000$

This is exactly as in the previous subtask, except that we now exploit that we are allowed to announce up to $t_a = 5p_a$, which allows us to increase the distance between the torches by a factor of 5.

Subtask 5. $N \le 100\,000$

The solution to this subtask interpolates between the solution to Subtask 3 presented above and the first full solution discussed below to achieve $t_a = p_a$ with at most $O(\sqrt[3]{N})$ fires.

Subtask 6. $N \le 500\,000$

Again, this can be solved by spreading out the fires from the previous solution by a factor of 5.

Subtask 7. No further constraints.

There are at least two different approaches that yield full score:

First full solution. Having blocks of a fixed size will of course only ever get us that far, so let us combine this with the idea of having the distances between the fires grow exponentially as we walk from right to left. A first attempt to implement this strategy might be to always have for any k a flaming torch at the largest multiple of 2^k that is $\le n$. Unfortunately, this does not work for similar reasons as in the third subtask: when n itself is a power of 2, then reducing n by 1 will still incur a large cost. Fortunately, also the solution to this problem is similar (although the analysis is somewhat harder this time): we enforce flaming torches at the *two* largest multiples of 2^k for each k, and up to one additional torch in the penultimate block of size 2^k to light the third largest multiple of 2^{k-1} if this becomes necessary during *leave*.

Second full solution. Let us analyze the solution to the very first subtask more closely, for which we again flip our indexing. We have seen that we can handle a single call to *leave* if and only if $f_i \le 2f_{i-1} + 1$ for all i and the leftmost torch is on fire. Moreover, we can trivially maintain this invariant for any call to *join*.

The crucial observation now is that we can also uphold this invariant for any call to *leave* by announcing p_a and then selecting fires as follows: assume we have already selected new torches $1 = g_1 < g_2 < \cdots < g_i$ satisfying the above; to select the next torch, we have to show that one of the torches $g_i + 1, \dots, 2g_i + 1$ is on fire. This is obviously true if one of these torches is the leftmost one or if $g_i < p_a$. In the remaining case, consider the smallest j with $f_{j+1} - 2p_a > 2g_i + 1$ (note that $f_{j+1} - 2p_a$ is the rightmost torch lit by the performer that was previously at position f_{j+1}). Then $f_j - 2p_a < 2g_i + 1$ by minimality, but also

$$f_j - p_a \ge \frac{f_{j+1} - 1}{2} - p_a > g_i.$$

Thus, at least one of the torches $f_j - 2p_a, \dots, f_j - p_a$ (all of which are currently on fire) lies in $\{g_i + 1, \dots, 2g_i + 1\}$ as claimed.

We now have to be slightly clever in selecting the g_i's in order to ensure that our program runs in time and that there aren't too many torches on fire.

- One way to do so is to always pick the largest g_{i+1} possible, which guarantees that $g_{i+2} > 2g_i + 1$ and hence ensures that we only ever have at most $2 \log_2 N + \epsilon$ torches on fire.
- More generally, you can come up with any reasonably time-efficient way to construct the g_i's and then 'sparsify' your result using the previous observation. For example, one uniform way that avoids the above case distinction is to start with the values $f_j - p_a$ plus the rightmost performer and then double each of them until you would run past one of the previously constructed elements.

Partial solutions. While we only described the full solutions above, there are various ways to obtain partial scores on the last subtask. One way to do so is to use larger bases in the above construction, i.e. only guaranteeing that $f_i \leq \alpha f_{i-1} + 1$ for some $\alpha > 1$. This becomes relevant if one uses less clever ways to sparsify the fires, which would otherwise result in too many torches being on fire.

In addition, there are several ad-hoc constructions.

Bring Down the ~~Sky~~ Grading Server (grading-server)

by Tobias Lenz and Lucas Schwebler

We write N for the maximum allowed value of c_H, f_H, c_G, f_G. Moreover, we say that a player *sabotages* the other player if they take down one of their firewalls.

Subtask 1. $S, c_H, f_H, c_G, f_G \leq 75$

We can represent every possible state of the game with a tuple of four integers (c_1, f_1, c_2, f_2), where (c_1, f_1) are the computing power and number of firewalls of the current player and (c_2, f_2) are the same for the other player. There are $O(N^4)$ such states, and so we use DP to compute whether each state is a winning or losing state for the first player. This works using the standard observation that a state is winning if and only if it can reach a losing state.

Subtask 2. $S, c_H, f_H, c_G, f_G \leq 300$

Lemma 1. *If (c_1, f_1, c_2, f_2) is winning for player 1, then the state is still winning if we increase c_1 or f_1. In the same way, if the initial state is losing, it stays losing if we increase c_2 or f_2.* □

Thus, we can compute $C(f_1, c_2, f_2)$ as the minimum c_1 such that (c_1, f_1, c_2, f_2) is a winning state. This can be done with DP in $O(N^3 \log N)$ with binary search. It's also possible to compute it in $O(N^3)$ using $C(f_1, c_2, f_2) \leq C(f_1, c_2 + 1, f_2)$.

Implementation detail. One has to be careful when implementing this. Otherwise, the DP might get cyclic dependencies when player 1 attacks. This can be solved by also trying both possible moves for player 2 after an attack by player 1.

Subtask 3. $S = 1$

Define $\alpha_1 := c_1 - S \cdot f_2$ and $\alpha_2 := c_2 - S \cdot f_1$. Intuitively, those values represent the damage caused by a player if they attack the other. Obviously, if $\alpha_1 \leq 0$, the only sensible choice for player 1 is to sabotage. In this interpretation, taking down a firewall just increases α_1 by S and this move can be preformed at most f_2 times. Thus, increasing f_2, for a fixed α_1, makes a state better for player 1, i.e. if $(\alpha_1, f_1, \alpha_2, f_2)$ is a winning state, so is $(\alpha_1, f_1, \alpha_2, f_2 + 1)$.

Lemma 2. *If $\alpha_1 \geq S$ and $\alpha_2 \leq S$, there is a winning strategy for player 1.*

Proof. Player 1 can maintain this invariant by attacking: Afterwards the new value of α_2 is $\alpha_2 - \alpha_1 \leq S - S = 0$. Thus, player 2 has to sabotage. Then, we still have $\alpha_2 \leq S$, and α_1 remains unchanged. □

Lemma 3. *If $\alpha_2 \geq S$, it is optimal for player 1 to attack.*

Proof. Assume for contradiction this were not the case and pick a $(\alpha_1, f_1, \alpha_2, f_2)$ with $\alpha_2 \geq S$ and f_2 minimal, such that sabotaging is a winning move, but attacking is not. In particular, $f_2 \geq 1$.

Suppose that player 1 sabotages, increasing α_1 by S. If player 2 attacks, this will decrease α_1 back to a value $\alpha_1' \leq \alpha_1$. As sabotaging was assumed to be a winning move, the new position $(\alpha_1', f_1, \alpha_2, f_2 - 1)$ is a winning position, hence so is $(\alpha_1, f_1, \alpha_2, f_2 - 1)$. By minimality of our counterexample this means that attacking is a winning move in this position, i.e. $(\alpha_2, f_2 - 1, \alpha_1, f_1)$ is losing. But then also $(\alpha_2, f_2, \alpha_1, f_1)$ has to be losing by the above observation, i.e. attacking was a winning move in the original position. □

Combining those observations with $S = 1$, we can obtain an optimal strategy for player 1. It turns out that whether a state is winning does not depend on f_1, f_2. Let $w(\alpha_1, \alpha_2)$ be **true** if player 1 wins and **false** otherwise. Using the above observations, we can calculate w recursively:

$$w(\alpha_1, \alpha_2) = \begin{cases} \neg w(\alpha_2 - \alpha_1, \alpha_1) & \text{if } \alpha_1, \alpha_2 > 0, \\ \textbf{true} & \text{if } \alpha_1 \geq \alpha_2, \\ \textbf{false} & \text{otherwise.} \end{cases}$$

Evaluating this formula recursively has running time $O(\log N)$ because we repeatedly subtract one value from the other, similar to the Euclidean Algorithm, until one of them becomes ≤ 0.

Subtask 4. $S, c_H, f_H, c_G, f_G \leq 2\,000$

We can reuse Lemma 2 and 3. This way, we know the optimal strategy if $\alpha_1 \leq 0$, $\alpha_1 \geq S$ or $\alpha_2 \geq S$. Notice that if $f_2 > 0$ and $\alpha_2 \leq 0$, player 1 can sabotage and win with Lemma 2. Thus, the only cases in which we don't know the optimal strategy are those which satisfy

$$0 < \alpha_1, \alpha_2 < S, \quad f_2 > 0, \quad \text{and} \quad f_1, f_2 \leq \frac{N}{S}.$$

The last inequality must hold because otherwise at least one α_i would be negative. We can compute a DP for these unknown states to determine which player wins. There are $O(N^2)$ many such states and each reduction takes $O(\log N)$ by a similar analysis as above.

Subtask 5. $S \leq 400$

Suppose that the game is an interesting state, so we don't know the optimal move yet. If player 1 sabotages, this increases α_1 to a value $> S$. Thus, player 2 has to attack. In total, this increases α_1 by $\beta_2 := S - \alpha_2$. Define $\beta_1 := S - \alpha_1$ similarly.

Lemma 4 (Death by sabotage)**.** *Player 1 has a winning strategy if $\beta_2 \cdot f_2 \geq \beta_1$.*

Proof. Player 1 sabotages f_2 times in a row. This increases α_1 by $f_2 \cdot \beta_2$. If $\beta_2 \cdot f_2 \geq \beta_1$, then $\alpha_1 + \beta_2 \cdot f_2 \geq S$ and so player 1 wins by Lemma 2. □

Now all interesting states satisfy $\beta_2 \cdot f_2 < \beta_1 < S$. Before attacking, player 1 needs to make sure that player 2 cannot apply the same strategy to win. After the attack of player 1, α_2 is at least $-S$. If $\beta_1 \cdot f_1 \geq 2S$, player 2 can make α_2 larger than S—and therefore win—by taking down all firewalls. So player 1 must not attack if $\beta_1 \cdot f_1 \geq 2S$.

Lemma 5. *There are $O(S^2 \log S)$ interesting states.*

Proof. Using the standard approximation of the harmonic series we compute the number of interesting states as

$$\sum_{\beta_1, \beta_2, f_1, f_2} [\beta_1 f_1 \leq 2S][\beta_2 f_2 \leq \beta_1] \leq \sum_{\beta_1} \sum_{\beta_2} \frac{2S}{\beta_1} \cdot \frac{\beta_1}{\beta_2} = O\left(\sum_{\beta_1} 2S \log \beta_1\right) = O(S^2 \log S).$$ □

Subtask 6. $f_H, f_G \leq 125$

We now present a solution which is fast when f_1, f_2 are small.

Lemma 6. *For fixed f_1, f_2 there exists a **critical attack value** $\gamma := \gamma(f_1, f_2)$ such that it is optimal for player 1 to attack if $\alpha_2 \geq \gamma$ and optimal to sabotage otherwise.*

Proof. Consider the states $T_1 = (\alpha_1, f_1, \alpha_2, f_2)$ and $T_2 = (\alpha_1 + 1, f_1, \alpha_2 + 1, f_2)$.

- If attacking is a winning move in T_1, it is also a winning move in T_2: Indeed, after the attack we get the states
$$T_1' = (\alpha_2 - \alpha_1, f_2, \alpha_1, f_1) \quad \text{and} \quad T_2' = (\alpha_2 - \alpha_1, f_2, \alpha_1 + 1, f_1).$$
Those states are identical except for the value of α_1. Since it's higher in T_2', the latter state is still winning for player 1.
- If sabotaging is a losing move in T_1 it is also in T_2: Indeed, after sabotaging and the necessary following attack by player 2, we are in the states
$$T_1' = (\alpha_1 + S - \alpha_2, f_1, \alpha_2, f_2 - 1) \quad \text{and} \quad T_2' = (\alpha_1 + S - \alpha_2, f_1, \alpha_2 + 1, f_2 - 1).$$

For fixed f_1, f_2 let now $h_A(\alpha_2)$ denote the minimum α_1 with which player 1 wins if he attacks in the state $(\alpha_1, f_1, \alpha_2, f_2)$, and let $h_F(\alpha_2)$ be the same if he sabotages. From the above points it follows that

$$h_A(\alpha_2 + 1) \leq h_A(\alpha_2) + 1 \quad \text{and} \quad h_F(\alpha_2 + 1) \geq h_F(\alpha_2) + 1,$$

hence $h_F(\alpha_2 + 1) - h_A(\alpha_2 + 1) \geq h_F(\alpha_2) - h_A(\alpha_2)$. So the difference $\delta(\alpha_2) := h_F(\alpha_2) - h_A(\alpha_2)$ is increasing in α_2. Noticing that attacking is an optimal move for every α_1 if and only if $\delta(\alpha_2) \geq 0$ finishes the proof. □

Observe that once we know all the critical attack values, we can simply simulate the game completely to determine the winner, which in turn allows us to compute all the $\gamma(f_1, f_2)$ recursively. To compute $\gamma(f_1, f_2)$, we use binary search to find minimum α_2 with $h_A(\alpha_2) \leq h_F(\alpha_2)$. Computing h_A, h_F is also done with binary search.

However, to make the simulation efficient enough, we will have to do several sabotages in one step, which requires us to find for a given state the largest $f_2' \leq f_2$ such that attacking is optimal. This can be done using binary search on a segment tree. A somewhat intricate analysis* then reveals that our simulation only takes $O(\log \log S)$ rounds, leading to a runtime of $O\big((Q + F^2 \log^2 F) \log N \log \log S\big)$ where F denotes the maximum value of f_1, f_2.

Subtask 7. No further constraints

The winning strategy from subtask 5 tries to take down all firewalls. Let's try to take down x firewalls, let player 2 do something, and take down the remaining $f_2 - x$ firewalls in the next step. Suppose that the current state is interesting in the sense of subtask 5. If player 1 attacks, player 2 cannot increase the value of α_2 to S (otherwise the state would not be interesting). If player 1 takes down x firewalls before attacking, his new β_1' equals $\beta_1 - \beta_2 x$. Thus, even if player 2 takes down all firewalls, he will have $\alpha_2 \leq S - \beta_2 x f_1$ because every sabotage will increase α_2 by β_2' instead of β_2. This means $\beta_2' \geq \beta_2 x f_1$. Now, player 1 takes down all remaining firewalls, increasing α_1 by at least $\beta_2 f_1 x(f_2 - x)$. If we now specialize to $x := \frac{1}{2} f_2$, this leads to an increase of $\frac{1}{4}\beta_2 f_1 f_2^2$. Since $\alpha_1 > -S$ before player 1 takes down the remaining firewalls, we see that this gives player 1 a winning strategy if

$$\beta_2 f_1 f_2^2 > 8S.$$

Note that in case of $f_1 = 0$ player 1 can still apply the same strategy if $\beta_2 f_2^2 > 8S$, while the case with $f_2 = 0$ is not interesting because player 1 has to sabotage. So all interesting states satisfy $f_2 > 0$ and $\beta_2 \max\{f_1, 1\} f_2^2 \leq 8S$.

* An important step in this analysis is to show that if player 1 attacks, player 2 has to sabotage until $f_1' \leq \frac{f_1}{f_2}$.

Lemma 7. *There are $O(S \log S)$ tuples (f_1, β_2, f_2) with $f_2 > 0$ satisfying $\beta_2 \max\{f_1, 1\} f_2^2 \le 8S$.*

Proof. For any $p \ge 0$, there are

$$\sum_{f_2=1}^{p} \lfloor p / f_2^2 \rfloor \le p \sum_{f_2=1}^{\infty} f_2^{-2} < 2p$$

pairs (f_1, f_2) with $1 \le f_1, f_2$ and $f_1 \cdot f_2^2 \le p$ because the sum $\sum_{i=1}^{\infty} i^{-2}$ converges to $\frac{\pi^2}{6} < 2$. Taking $p = 8S/\beta_2$ and summing over all β_2 then gives the desired bound. □

For every such tuple, we compute the minimum c_1 with which the state is winning with binary search; this gives us an $O(S \log^2 S)$ solution. Again, one needs to implement this carefully to prevent cyclic DP dependencies.

Final remarks.

Combining the solutions for the last two subtasks with some additional ideas, it is also possible to solve this problem in time $O\left(\sqrt{S} \log^4 S + Q\sqrt{S} \log S \log\log S\right)$. Suppose that we are in an interesting state and player 1 attacks. This implies that we had $f_2 < \beta_1$ (otherwise, use death by sabotage). After the attack, player 2 sabotages some rounds until he gets into an interesting state. Then, $f_1^2 f_2^2 < f_1^2 f_2 \beta_1 < 8S$. Notice that there are only $O(\sqrt{S} \log S)$ such pairs (f_1, f_2). Use the idea of subtak 6 for them, leading to a precomputation time of $O(\sqrt{S} \log^4 S)$. To answer a query, we simply try all possible number of sabotages before player 1 attacks. For every of those number of sabotages, we can find the winner very efficiently using the idea from subtask 6. Since $f_2 \le \sqrt{8S}$, the number of sabotage rounds to try is quite small and we get the above time complexity. This is fast enough to solve the problem for constraints $S \le 10^6, Q \le 25\,000$.

Brought Down the Grading Server? (balance)

by Lukas Michel

We say that the submissions to a task are balanced if the maximum and minimum number of submissions for this task during the rejudging differ by at most one.

Subtask 1. $S = 2$ and $N, T \leq 20$

In this subtask, we can simply enumerate all possible ordered assignments and output one for which the submissions to all tasks are balanced. This can be implemented in time $O(2^N \cdot (N + T))$.

Subtask 2. $S = 2$

First, consider the case where the total number of submissions to each task is divisible by 2. In fact, in this case we can assume that every task has exactly 2 submissions: we can simply replace a task with s submissions by $s/2$ tasks with 2 submissions each. If the number of submissions to each of the new tasks is balanced, this is also true when we replace them again with the initial task.

Now we have to make sure that the two submissions to each task and the two submissions in the list of each core are assigned to different timeslots. Note that these submissions form cycles of submissions that are pairwise either to the same task or in the list of the same core. Once we assign one of these submissions to a timeslot, this means that the other submission of the corresponding task and the corresponding core have to be assigned to the other timeslot. This, in turn, implies that two other submissions have to be assigned to the same timeslot as the initial submission, and so on.

This means that once we assign a single submission of a cycle to a timeslot, this determines uniquely to which timeslots all other submissions of this cycle have to be assigned to. Moreover, by construction, the two submissions to each task are assigned to different timeslots, and so this ensures that the submissions to all tasks are balanced, as required. This can be implemented in time $O(N + T)$, solving the first group in this subtask.

If there are tasks whose number of submissions is not divisible by 2, we can replace them by multiple tasks with 2 submissions each and one task with 1 submission. Then, in addition to cycles, we will also have paths, but the same construction also applies to them. This can still be implemented in time $O(N + T)$, and solves the entire subtask.

Subtask 3. $N \cdot S \leq 10\,000$

From this subtask on, we frame the problem in the terms of graph theory. More precisely, we construct a bipartite graph with the left vertices being the cores, the right vertices being the tasks, and the submissions being edges between them.

Again, we will first focus on the case where the number of submissions to each task is divisible by S, or equivalently where the degree of every right vertex of the bipartite graph is divisible by S. As before, we can assume that every degree is exactly S by splitting a vertex of degree d into d/S vertices with degree S each.

Now, for each minute, our assignment has to pick exactly one submission from the list of every core while also selecting exactly one submission to each task. In our bipartite graph, such a set of submissions corresponds to a perfect matching. Fortunately, it is known that every regular bipartite graph—that is, a bipartite graph where the degrees of all vertices are the same—has a perfect matching. This can be proven using Hall's Theorem, for example.

From this, we get the following algorithm: first, compute a perfect matching in the bipartite graph, and let these be the submissions evaluated by the cores in the first minute. Then, remove those edges from the graph. The resulting bipartite graph is still regular, so we still know that it contains a perfect matching. Therefore, we can repeatedly compute perfect matchings and remove them from the graph until we have a set of submissions evaluated by the cores for every minute.

Since the bipartite graph has N left vertices and $N \cdot S$ edges, there are simple matching algorithms that run in time $O(N^2 \cdot S)$. This yields an overall runtime of $O(N^2 \cdot S^2)$, solving the first group in this subtask.

Similar to before, if the degrees of some right vertices are not divisible by S, we can replace them by multiple vertices with degree S each and one vertex with degree in $[1, S-1]$. However, in this bipartite graph, simply finding complete left-to-right matchings and removing them repeatedly might not work as we have no guarantee that such matchings will always exist.

Instead, we can transform our graph into a regular bipartite graph by adding new left vertices and connecting them to right vertices with degree lower than S. If we pick perfect matchings in this regular bipartite graph, they reduce to complete left-to-right matchings in the original graph, as required. The regular bipartite graph will have at most $N + T$ left vertices, so the previous algorithm runs in time $O((N+T)^2 \cdot S^2)$, solving the second group of this subtask.

Finally, if $T \geq N$, we can observe that before constructing the regular bipartite graph, we can simply merge any two right vertices if the sum of their degrees is at most S. Once such merges are no longer possible, we will have $T \leq 2N$, and so we can apply the previous algorithm. This solves the entire subtask with a runtime of $O(N^2 \cdot S^2)$.

Subtask 4. No further constraints.

In this subtask, instead of removing matchings one-by-one, we will employ a divide-and-conquer approach: we want to split the edges of the bipartite graph into two sets such that for every vertex (both left and right), half of its incident edges, up to rounding, are in each set. Since S is a power of two, we can apply this recursively, and this will produce the required balanced ordered assignment: this splits the edges incident to left vertices equally, so every core will evaluate exactly one submission per minute, and it also splits the edges incident to right vertices equally, which means that the submissions to any task will be balanced in the end.

To split the edges of the bipartite graph into two such sets, we can use Euler tours. First, if the degree of every vertex is even, we can take any Euler tour. Then, we put every second edge into one set and every other edge into the other set. Since consecutive edges are in different sets, this ensures that exactly half of the edges incident to any vertex end up in the first set and half of the edges end up in the second set, as required. If every degree was divisible by S in the beginning, the degree of every vertex will stay even throughout this process, and so this solves the first group of this subtask. Since Euler tours can be computed in linear time, this gives a runtime of $O(N \cdot S \log S)$.

Finally, whenever some right vertices have odd degree, we can add a new left vertex connected to all of these right vertices with odd degree. Then, every vertex will have an even degree, and we can apply the approach from before to partition the edges into two sets. If we remove the additional left vertex, this still guarantees that for every vertex, up to rounding, half of its incident edges are in each set. This solves the entire task, with a runtime of $O(N \cdot S \log S)$.

Final remarks.

The task could also be solved if S is not a power of two. For this, we can combine the ideas of Subtasks 3 and 4. If S is odd, we can remove a perfect matching from the graph, and if S is even, we can split the edges into two sets as above. Overall, this would give a runtime of $O(N^2 \cdot S \log S)$, or better if the matching algorithm is more efficient. However, this would be more annoying to implement, and it was difficult to prevent efficiently implemented matching solutions to subtask 3 from solving these testcases as well, which is why such a subtask was not included in this problem.

Tricks of the Trade (trade)

by Lukas Michel and Tobias Lenz

Subtask 1. $N \leq 200$

In this subtask we have enough time to check each possible segment by computing the sum of the K largest sale values and subtracting from this the sum of the costs of the segment. We can implement this check in $O(N)$ by using the *nth_element* function, but even computing it via sorting in $O(N \log N)$ is fine. Moreover, we can also compute the optimal indices by marking those elements that are at least as large as the K-th largest element of each such segment, where we reset all markings whenever we find a new segment with a higher profit.

Overall the runtime of this solution is $O(N^3)$ or $O(N^3 \log N)$.

Subtask 2. $N \leq 6\,000$

In this subtask we improve the above idea by handling each segment in $O(\log N)$ time. For this, we fix the left endpoint of the segment, and we iterate through every possible right endpoint. At the same time, we keep a priority queue with the K largest elements of the segment, and we also keep track of the sum of these elements as well as the costs of the current segment. This allows us to compute the profit of the current segment in time $O(\log N)$.

To compute the optimal indices, for each segment with the maximum profit that we encounter we can store the segment along with its K-th largest element. We will denote the K-th largest element of segment $[\ell, r]$ by $t_{[\ell,r]}$. An index i is then optimal if and only if there is a maximum profit segment $[\ell, r]$ with $\ell \leq i \leq r$ and $s_i \geq t_{[\ell,r]}$. Computing all such indices can be done efficiently with a minimum segment tree or a sweep line approach, both in time $O(N + S \log N)$ where S is the number of segments with maximum profit.

Overall this leads to a runtime of $O(N^2 \log N)$.

Subtask 3. $K = 2$

For $K = 2$, we observe that in any maximum profit segment $[\ell, r]$ you have to sell the robots ℓ and r to the other contestants.

This allows us to iterate through all possible right endpoints r while we maintain the maximum profit that we can gain from picking any left endpoint $\ell < r$. This profit is $s_r + m_r$ where $m_r := \max_{1 \leq \ell < r} s_\ell - \sum_{i=\ell}^{r} c_i$. We can update m_r in $O(1)$ whenever we move one step to the right by noting that

$$m_r = \max\{m_{r-1} - c_r, s_{r-1} - c_{r-1} - c_r\}.$$

This allows us to compute the maximum profit in $O(N)$.

To compute the optimal indices, we can simply store all right endpoints that are part of a maximum profit segment and then repeat the procedure backwards to obtain all left endpoints of maximum profit segments. Together, these two sets form the set of optimal indices by our observation.

Subtask 4. $K \leq 200$

Let $p(r, k)$ denote the maximum achievable profit if we buy some segment $[\ell, r]$ and sell k robots of this segment. Then, the maximum profit is $\max_{1 \leq r \leq N} p(r, K)$. As base cases, we have $p(r, 0) = 0$ and $p(0, k) = -\infty$ for $k > 0$. Then, we can calculate p recursively for $r, k > 0$ as

$$p(r, k) = \max\{p(r-1, k), p(r-1, k-1) + s_i\} - c_i.$$

This is because we can choose whether or not to sell the i-th robot. This formula can be evaluated in $O(NK)$ using DP.

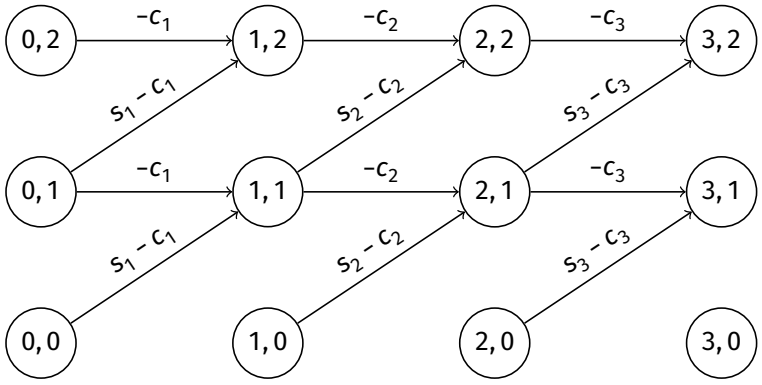
To find the optimal indices, we reconstruct all optimal DP transitions. If a transition $(i, k) \to (i + 1, k + 1)$ appears in an optimal solution, then the i-th robot is part of an optimal transaction.

Subtask 5. No further constraints.

We can visualize the DP of the previous subtask as a directed acyclic graph with nodes (r, k) and edges

- $(i - 1, k) \to (i, k)$ with weight $-c_i$ *and*
- $(i - 1, k) \to (i, k + 1)$ with weight $s_i - c_i$.

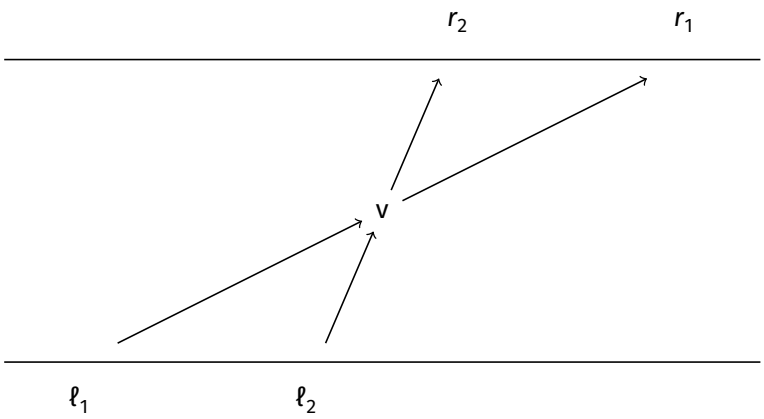
The problem of finding the maximum profit is then equivalent to finding the longest path from some node of the form $(\ell, 0)$ to some node (r, K) in this graph. For example, the graph for $N = 3$ and $K = 2$ looks like this:



For a left endpoint ℓ, we say that a right endpoint $r > \ell$ is ℓ-*optimal* if buying the segment $[\ell, r]$ achieves the maximum profit possible for this fixed left endpoint ℓ.

Lemma 1. *Let $\ell_1 < \ell_2 < r_2 < r_1$ be such that r_1 and r_2 are ℓ_1- and ℓ_2-optimal respectively. Then, r_2 is also ℓ_1-optimal, and r_1 is also ℓ_2-optimal.*

Proof. Let p_1 and p_2 be longest paths corresponding to the intervals $[\ell_1, r_1]$ and $[\ell_2, r_2]$. The paths p_1 and p_2 must intersect at some vertex v of the graph:



The paths from v to r_1 and from v to r_2 must have the same length as we could otherwise replace the part of p_1 or p_2 that comes after v with a longer path. In particular, the path from ℓ_1 to r_2 via v has the same length as p_1, and the same holds for the path from ℓ_2 to r_1 compared to p_2. Hence, buying the segment $[\ell_1, r_2]$ or $[\ell_2, r_1]$ also achieves the maximum profit. □

Let $opt(\ell)$ be the smallest ℓ-optimal right endpoint. If we can compute $opt(\ell)$ for every possible left endpoint ℓ, then the maximum overall profit is the maximum profit of the segments $[\ell, opt(\ell)]$. From the lemma above we get that $opt(\ell) \leq opt(\ell + 1)$.

This means that we can apply the divide and conquer optimization: First, we compute $opt(m)$ for $m = N/2$ by iteratively testing every possible value r. Then, to compute $opt(\ell)$ for $\ell \in [1, m - 1]$, we only need to consider $r \leq opt(m)$ as possible right endpoints, and for $\ell \in [m + 1, N]$ we only check $r \geq opt(m)$. For these intervals, we can apply the idea recursively. In total, this means that we have to check at most $O(N \log N)$ values of r.

However, during this divide and conquer algorithm, we still need an efficient way to compute the maximum profit of a segment $[\ell, r]$. Since we can compute the costs of such a segment with prefix sums (or one of many other ways), we focus on efficiently computing the sum of the K largest elements of this segment. For this, recall our approach from Subtask 2. There, we kept a set of the K largest elements of our current segment $[\ell, r]$ as well as their sum, and we were able to efficiently add elements to this segment.

In our divide and conquer algorithm, we can now always move our current segment $[\ell, r]$ to the segment where we need to know the sum of the K largest elements. However, for this we would also need to be able to remove elements from the segment. Fortunately, this is also possible: in addition to the set with the K largest elements, we can also keep a set with all the other elements of the current segment. If we now delete an element, we can delete it from the appropriate set and rebalance the two sets so that afterwards the top set contains the K largest elements once more.

With the standard analysis of the divide and conquer optimization, we can prove that ℓ and r move at most $O(N \log N)$ steps during this process, and so this algorithm runs in time $O(N \log^2 N)$. It would also have been possible to use persistent segment trees to compute the sum of the K largest elements, which would have resulted in the same complexity.

For a full solution, it remains to compute the optimal indices. In Subtask 2 we already noted that if there are S segments with maximum profit, we could do it in time $O(N + S \log N)$. However, in this subtask, it might hold that $S \in \Theta(N^2)$ which makes this approach too slow. So, we have to reduce the number of segments that we need to consider.

For this, assume that $[\ell_1, r_1]$ and $[\ell_2, r_2]$ are segments with maximum profit with $\ell_1 < \ell_2 < r_2 < r_1$ and that $i \in [\ell_1, r_1]$ is an optimal index. This means that i is one of the K largest elements in one of these two segments. In this case, we know from the lemma that $[\ell_1, r_2]$ and $[\ell_2, r_1]$ are also segments with maximum profit. Since these segments are shorter, it follows that i must also be an optimal index of one of the segments $[\ell_1, r_2]$, $[\ell_2, r_2]$, or $[\ell_2, r_1]$.

In particular, if $\ell_1 < \ell_2$ are left endpoints of segments with maximum profit with no such endpoint in between, for ℓ_1 we only need to consider right endpoints r with $opt(\ell_1) \leq r \leq opt(\ell_2)$ when computing all optimal indices. This means that we will only consider at most $2N$ segments in total, allowing us to compute all optimal indices with a two pointer approach in $O(N \log N)$.

The Ties That Guide Us (incursion)

by Lukas Michel

Throughout, we will refer to the problem in graph theoretic terms: the floor plan describes a tree with N nodes, such that any node has degree ≤ 3.

Subtask 1. No degree 3 node

In this subtask, the tree is actually a line. Assume first that the number N of nodes is odd. While you and your assistant might receive this line in different numberings, there is one special node that can be identified without reference to any numbering, namely the node in the middle. This suggests a two step procedure to get to the secret node: first, we walk to the midpoint, and then we have our assistant (who would not know our starting point) guide us from there—for this, he can simply mark the path from the midpoint to the secret node (by placing exactly one tie in the respective rooms and no ties anywhere else).

Note that if our starting position lies on the opposite side of the midpoint than the secret node, this will need at most two steps over the shortest path between them (namely, when we 'overshoot' and have to walk back to the secret node). However, when we start on the same side as the secret node, actually walking to the midpoint might result in a huge detour. This issue can be solved as follows:

- If our starting node is marked, we simply walk *away* from the midpoint (again, overshooting by at most 1, resulting in a maximum detour of 2).
- If not, we can simply stop at the first marked node we encounter (resulting in no detour at all).

The case where N is even is only slightly more complicated: instead of a unique midpoint, we now have two (neighbouring) nodes in the middle. In the first step, our assistant will simply mark all nodes on the path from the secret node to the node in the middle that is *closer*. Almost the same strategy as above will then work for us to find the secret node where we now try to walk to the marked point in the middle:

- If our current node is marked we walk towards the end that is closer to us; the last marked node we encounter is the secret node.
- If not, we walk to the midpoint that is *further* from our starting location. If we encounter a marked node on the way, the first such node has to be the secret node. Otherwise we follow the markings left by our assistant; the last marked node will be the secret one.

Subtask 2. Precisely one degree 2 node

In this subtask, the tree no longer needs to be a line (and in fact, it will almost never be). However, there is still one special node that both we and our assistant can identify: the unique node with degree 2. Using this observation we can now use a very similar strategy to the line case:

- Our assistant will mark the path from the unique degree 2 node to the secret node.
- We first walk from our current node to the unique degree 2 node until we visit a marked node for the first time.
- At that point, we can follow the markings (moving away from the unique degree 2 node) to the secret node.

However, the third step is actually more subtle this time: as we are no longer just dealing with a line, there might be several possibilites for the next node, and we have no way to tell which one of these is

marked without actually walking there. However, whenever we walk to the wrong node, this incurs a cost of 2, so we can't allow ourselves to have this happen too often.

To solve this, root the tree at the unique degree 2 node. We will then always walk into the *larger* of the two subtrees first: if this is correct, everything is fine, and if not, then the marked path has to continue to the *smaller subtree*. Thus, whenever we make a mistake, this halves the size of the remaining subtree, which can only happen $O(\log N)$ times. Let us analyze this more closely to see that it really fits into the task constraints, for which it will be useful to argue in a bottom-up fashion instead:

- Let k denote the number of children of our secret node (note that k is at most 2 except for the easy case that the secret node agrees with our chosen root). Then we might make a detour of up to $2k$ at this node because of overshooting.
- Consider now the marked nodes $v_1, \dots, v_\ell$ apart from the secret node where we we went to the wrong subtree first (numbered from bottom to top), and let s_i denote the corresponding subtree sizes; we moreover write s_0 for the size of the subtree rooted at the secret node. As we always go to the larger subtree first, we have $s_{i+1} \geq 2s_i + 1$. Combining this with $s_0 \geq k + 1$, we see inductively that $s_\ell \geq 2^\ell(k + 2) - 1$.
- Our total detour is $2(k + \ell)$. Trying $k = 0, 1, 2$, we see that for a total detour of 32 we would need to have $N \geq s_\ell \geq 2^{16} - 1 = 65\,535$ (achieved for $k = 2, \ell = 14$), which is larger than the maximum number of rooms.

Subtask 3. No further constraints

In the final subtask, we are given a completely general tree. In order to adapt the solution to the previous subtask to this case, we somehow need to find a special node again that can be identified without referring to the numbering of our nodes and edges (or, to use fancy graph theoretic terminology, a node fixed by any automorphism of our tree). While already the line case shows that this need not be possible, we can again solve this by also allowing a set of two neighbouring nodes instead. Possible such choices are then the *centers* of the tree (nodes minimizing the maximum distance to any other node) or the *centroids* (nodes minimizing the maximum component size when we remove them). We can now again simply have our assistant mark the path from the secret node to the special node (or the closer one of the two special nodes, if there is more than one) and then use the same strategy as before to locate the secret node.

In our analysis, we need to consider one extra case: the root might have degree 3. In this case, if the subtree containing the marked path is the smallest, we might actually make a detour of 4 right at the root. However, in order to make a detour of 28 in the subtree of the marked path, this would then need to contain at least $2^{14} - 1 = 16\,383$ nodes (recycling the analysis from the previous subtask), resulting in $N \geq 49\,150$.

Note in particular that the bounds in this task are quite tight: already with the slightly weaker bound $N \leq 50\,000$ our strategy would not be able to guarantee a detour of 30. In fact, one can force *any* strategy to make a detour of at least 32 on the following tree with $N = 49\,150$ nodes: take three balanced binary trees of height 13 and add a new node connected precisely to their roots. This is actually the only instance with less than 65 535 nodes on which our strategy will lead to a detour larger than 30.

Solutions with more ties.

If we are allowed to mark the nodes with integers up to $N - 1$ (i.e. have our assistant leave behind up to $N - 1$ ties per room), a natural strategy is to simply mark each node with the distance to the special node. We will then immediately recognize the special node upon entering, and before that we can always simply walk to the unique neighbour whose marking is smaller than the current one. This solves the line case directly, while for the general case we again need the idea to check the larger subtree first.

For a solution with markings bounded by 2, we observe that the distance can only increase or decrease by 1 when we travel to a neighbouring node. Thus, to recognize whether the distance decreased, it is enough to remember the distance modulo 3. The secret node can be identified from this data as the unique node such that we cannot decrease the distance by walking to a neighbouring node.

How to Avoid Disqualification in 75 Easy Steps (avoid)

by Tobias Lenz

Throughout, we denote the positions of the chairmen by $1 \le a \le b \le 1\,000$, and $N = 1\,000$ is the number of positions.

Subtask 1. $R = 10$, $H = 1$, and both chairmen are located at the same position.

This subtask is easiest to solve if we invert the problem: instead of determining for each robot which positions it scouts, we will determine for each position p the set of robots $R_p \subseteq \{1, \dots, 10\}$ scouting it.

If both chairmen are located at position a, a robot r will detect at least one chairman if and only if $r \in R_a$. Hence, the results of all 10 robots are equivalent to knowing the set R_a. To determine a from this, we therefore need to ensure that the sets R_p are all different.

Fortunately, $2^{10} \ge 1\,000$, and so we can assign a unique subset $R_p \subseteq \{1, \dots, 10\}$ of the robots to each position p. A simple way to do this is to use the binary representation of p—then, the results of the robots spell out the position of the chairmen in binary, solving this subtask.

Subtask 2. $R = H = 20$

This subtask can be solved with two binary searches. With the first binary search, we want to determine the smallest position a of one of the chairmen. For this, we will always keep an interval $[\ell, r]$ that contains a, starting with $\ell = 1$ and $r = 1\,000$. Then, in each step, we pick $m = \lfloor (\ell + r)/2 \rfloor$, send a robot to the positions $\{\ell, \ell + 1, \dots, m\}$, and wait for its result. If this robot detects a chairman, we know that the interval $[\ell, m]$ must contain a, and so we set $r = m$. Otherwise, a must be contained in $[m + 1, r]$, and so we assign $\ell = m + 1$. Once we have $\ell = r$, which happens after at most $\lceil \log_2 1\,000 \rceil = 10$ steps (and 10 robots), we have determined a as required.

With the second binary search, we can simply determine the largest position b of one of the chairmen analogously, again with 10 robots. Therefore, we can determine the positions of both chairmen with 20 robots in total.

Note that we could also execute the above two binary searches simultaneously which would reduce the total time required from 20 hours to 10 hours. It is easy to prove by counting that at least 19 robots are necessary, but the Scientific Committee does not know whether there exists a solution that uses at most 19 robots.

Subtask 3. $R = 30$, $H = 2$

To find the positions of the two chairmen in 2 hours, recall our approach to Subtask 1. To all positions p, we had assigned a distinct set $R_p \subseteq \{1, \dots, 10\}$ of robots, namely the binary representation of p. We will denote the i-th bit of p by p_i, and so $i \in R_p$ if and only if $p_i = 1$.

Now, in this subtask, we will send 10 robots as described by the sets R_p, and we send an additional 10 robots for the complements R_p^c, all during the first hour. Based on the results of these robots, we will know for all i that either $a_i = b_i$, in which case we will also the value of a_i and b_i, or $a_i \ne b_i$.

Now, in the second hour, we need to reconstruct the remaining information. If i and j are bits such that both $a_i \ne b_i$ and $a_j \ne b_j$, we need to determine whether $a_i = a_j$, and equivalently $b_i = b_j$, or whether $a_i \ne a_j$, and equivalently $b_i \ne b_j$.

To do this, let i be a bit such that $a_i \ne b_i$. Now, for every j that is different between a and b, we will simultaneously send a robot to all those positions p where $p_i = p_j$. If this robot detects a chairman,

we know that $a_i = a_j$ and also $b_i = b_j$, and otherwise we know that $a_i \neq a_j$ and also $b_i \neq b_j$. This lets us reconstruct a and b based on the i-th bit. This strategy uses at most 29 robots in 2 hours.

Subtask 4. $R = 75, H = 1$

This is the only output-only subtask of this CEOI.*

Constructive solutions. We begin by describing some constructive solutions.

- Firstly, we can adapt the solution to Subtask 3. Instead of using the second phase of that strategy, we can, for each pair of bits i and j, already send a robot in the first phase to all positions p with $p_i = p_j$. This uses a total of $20 + \binom{10}{2} =$ **65** robots. Then, we already know the results for all robots that we would have sent out during the second phase, and so we can immediately reconstruct a and b after just one hour.
- To get a solution with fewer robots, we can use a base other than 2. We will describe a solution with base 3, where we again denote the i-th digit of a position p by p_i. Then, for every digit i and every $d \in \{0, 1, 2\}$, we send a robot to all positions p with $p_i = d$. Since there are $\lceil \log_3 1\,000 \rceil = 7$ digits, this uses 21 robots, and it will tell us for every digit i whether $a_i = b_i$, in which case we will also know the values of a_i and b_i, or whether $a_i \neq b_i$. In this second case we know which two digits a_i and b_i are, so we know two digits $d_i^1, d_i^2 \in \{0, 1, 2\}$ such that $\{a_i, b_i\} = \{d_i^1, d_i^2\}$, but we do not know whether $a_i = d_i^1$ or $a_i = d_i^2$, and similarly for b_i.

  Thus, for every pair of digits i and j we also need to send robots to make sure that if $d_i^1 \neq d_i^2$ and $d_j^1 \neq d_j^2$, we know whether d_i^1 and d_j^1 are both digits of a or both digits of b, or whether this is not the case. It turns out that it is again sufficient for this to send a robot to all positions p with $p_i = p_j$. Indeed, one of the digits d_i^1 or d_i^2 must have the same value as one of the digits d_j^1 or d_j^2, and so the result of this robot tells us whether these two digits are both digits of a or both digits of b. This uses an additional $\binom{7}{2} = 21$ robots, for a total of **42** robots.

  There also exists a solution with base 4 that uses **40** robots.
- Finally, it is possible to come up with a scheme that takes a solution for N positions and R robots and produces a scheme for N^2 positions and $3R$ robots. For this, let R_p be the sets of robots assigned to each position p in the scheme for N positions. We then represent the N^2 positions as pairs (p, q) with $1 \leq p, q \leq N$, and for (p, q) we send R robots each according to R_p, R_q, and $R_{p+q \pmod N}$. It can be checked that this allows you to uniquely determine the positions of the chairmen among all N^2 positions. Starting with a solution with $N = R = 6$ and applying this twice leads to a solution with **54** robots.

A general approach. Assuming that we have already determined where each robot is sent to, we can check efficiently whether this is a valid scheme that always determines the positions of the chairmen with certainty. To do so, consider again the sets R_p of robots assigned to each position p. A robot r will detect a chairman if and only if $r \in R_a$ or $r \in R_b$, or equivalently $r \in R_a \cup R_b$. This means that the results of the robots are equivalent to knowing $R_a \cup R_b$, and so we can determine a and b uniquely if and only if $R_a \cup R_b$ is different from all other such unions. This can be checked by representing each R_p in binary as an **__int128_t** so that $R_p \cup R_q$ is simply the bitwise or of the two numbers, running in time $O(N^2)$ when using a hash map to check for collisions.†

Reconstructing a and b from the answers then simply works in exactly the same way. We can iterate through all p and q and check whether the $R_p \cup R_q$ matches the results of all robots.

* Wait, what? Keep on reading to understand why...

† You might have noticed that no heuristic managed to get more points than intended in the last subtask...

All of our full solutions are based on generating the sets R_p *locally* on our own computer, potentially taking minutes or even hours, until we have a valid scheme. Such a scheme can then be encoded as a list of numbers in the submission, and we use the above strategy to determine a and b from the results of the robots.

Generating robots. Let us first describe some strategies that try to generate the plan robot by robot.

- Fix $0 < p < 1$ and send any robot ϱ to any position x with probability p, independently of any other choices. We claim that for sufficiently many robots this will yield a valid scouting plan with positive probability. To prove this, let us say that an assignment for a single robot *distinguishes* two sets $L := \{x, y\}$ and $L' := \{x', y'\}$ of possible chairman positions if the robot returns 1 for *precisely* one of these two sets. Note that this happens with a positive probability q (maximized for $p \approx \frac{1}{3}$). Moreover, for any other robot ϱ', the events that ϱ distinguishes the two given L, L' or that ϱ' distinguishes them are independent; in particular, for r robots the probability that we do not distinguish them is $(1 - q)^r$. Summing over all L, L' this yields an upper bound of $N^4(1 - q)^r$ for the probability that we do *not* find a correct scouting plan. For $r \to \infty$ robots, this failure probability converges to 0. In practice, this approach suffices to find solutions with **50** robots in reasonable time.
- The previous strategy tends to generate plans that almost work, i.e. there are few pairs (L, L') which we cannot separate. This allows us to get better results by 'supersampling': we first use the above strategy for $N > 1\,000$ positions ($N \approx 2\,000$ seems to be the sweet spot); as long as there are $\{x, y\}$ and $\{x', y'\}$ which we can't distinguish, we randomly throw away one of x, y, x', y'. This yields solutions with around **35** to **40** robots, depending on how much computing time you are willing to invest or how clever you are in selecting the positions to discard.
- Instead of all the fancy randomness, we can try to be more systematic in assigning positions to the robots, somewhat akin to binary search.

  For this, let us first consider the case of a single robot. Ideally, the two possible answers `0` and `1` should occur for around the same number of possible chairman positions. Doing the math, we see that for this we should send the robot to a fraction of

  $$\alpha := 1 - \frac{\sqrt{2}}{2} = 0.29289\ldots$$

  of the positions (note that this is *not* the optimal probability p from the random approach!).

  Similarly, for two robots, the outcomes `00`, `01`, `10`, and `11` should occur for around the same number of possible chairman positions. To achieve this, we can send the second robot to a fraction of α of the first robot's positions *and* to a fraction of α of the positions *not* visited by the first robot.

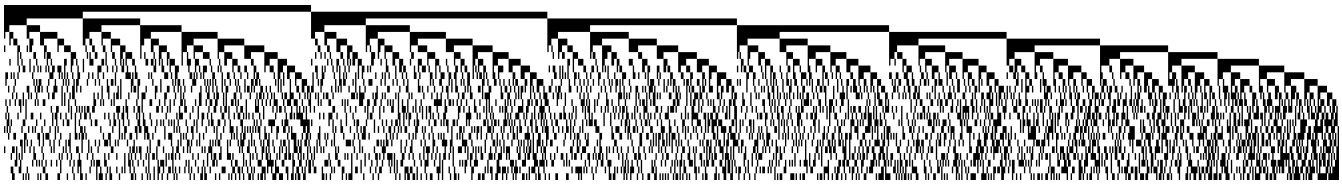
  If we iterate this idea to determine where to send k robots, we run into the issue that $\alpha^k \to 0$ rapidly; in particular, rounding errors take over around $k \approx 7$ and we do not separate the 'blocks' of possible positions for fixed return values evenly any more. We therefore switch to a greedy approach that tries several random plans for each individual robot, and then takes the one that minimizes the maximum size of any block. Possibly combining this with some local optimization, we obtain solutions with around **30** robots.

Generating positions. Another way to generate a solution is to fix the number of robots and to generate positions one-by-one. We start without any positions. Then, we can iteratively add new positions to our solution, always choosing a new set R_p for a new position p in such a way that there are no collisions with any or the previous positions.

- If we simply pick each set R_p randomly (with the above probabilities), this manages to generate a solution with **30** robots.
- If we try fewer robots than this, the number of valid choices for the next set R_p becomes very small. This means that every iteration takes very long and we do not reach the 1 000 positions that we need.

  Instead, at some appropriate point during the computation we can simply start to keep all possible compatible sets that would create no collision with the positions that we have generated to far. This makes it much faster to select the next set R_p. Moreover, we can connect this with other heuristics, such as selecting R_p in a way that maximises the number of remaining compatible sets. This can produce a solution with **27** robots.
- Right now, the Scientific Committee only knows a single way to obtain a solution with fewer robots. For this, we again select the sets R_p iteratively. However, we perform this in a very specific way. Namely, we only add sets with exactly 8 elements to our solution, and we also only add them if their symmetric difference with every set chosen so far contains at least 6 elements. Moreover, we try to add these sets in increasing order if we look at them in their binary representation. This produces a solution with 992 positions, which is not sufficient yet. If we then iterate through all remaining sets that we haven't tried so far (e.g. because they do not have the correct number of elements), and we try to add them greedily to our solution, it is possible to improve this solution to 998 positions. Unfortunately, this is still not enough. At this point we are stuck...

  ...is what we would have said if we had not tried to throw some simulated annealing on this solution to generate additional positions. After a short time, this manages to add two more positions to our solution. This is then a solution with **26** robots and 1 000 positions, shown in the picture below, where a black box (2 pixels wide and 10 pixels high) in row ϱ and column p means that robot ϱ is sent to position p:



  It turns out that our way of generating the initial solution with 992 positions is quite fragile. Whenever we tried to change any part of this approach (say by going through the sets R_p in a random order when trying to add them), the solution got much worse.

For 25 robots, the best we were able to do is a solution with slightly below 800 positions. The Scientific Committee also has no idea what the minimum number of robots is that a solution requires—we have not even been able to prove that more than 19 robots are necessary.