EGOI 2026 Editorial - Seating Plan

Task Authors: Tomohisa Hachiya, Takuma Masuda, Yuki Tanaka

Distances instead of people counts

Most of the calculations we'll need to make are nicer if instead of queries that return people counts in given ranges we consider each seat a point with an integer coordinate and have queries that return the distances between those points – i.e., a straight difference between the max and min of the three given coordinates. Therefore we'll immediately subtract 1 from each answer to the original query and we'll work with those values instead. The smallest possible distance is therefore 2 and the largest one is $N - 1$.

Brute force

Clearly, for $N \geq 5$, asking all $\binom{N}{3}$ possible questions is enough to determine the full answer. One way to determine the order after knowing all the answers is to deduce it as follows:

- The two guests who are involved in all questions with answer $N - 1$ are the ones sitting on the ends.
- We can arbitrarily assign one of them to seat 0 and the other to seat $N - 1$.
- There is exactly one question that involves the guest in seat 0 and has the answer 2.
- From this question we know the two guests X and Y in seats 1 and 2 but we don't know which is which.
- The question with guest X, guest in seat $N - 1$, and any unused guest will tell us whether X is in seat 1 or 2.
- Once we know the guests in seats 0 and 1, we can use all questions that involve them + a third guest to place everyone else.

Of course, in terms of having the simplest possible implementation, in the $N = 8$ case we can avoid all the casework. Instead, after asking all the questions we simply iterate over all possible permutations of guests until we find one consistent with all the answers we received.

For $N = 40$ we can still (barely) afford to ask all possible questions, we just need to use some smarter way of reconstructing the order (e.g., the one explained above), as examining $N!$ permutations is no longer feasible for $N = 40$.

Second subtask

We can start by making all queries of the form $(0, 1, x)$ to sort everyone according to their distance from the pair $(0, 1)$. In general, there will be some distances with two guests and then some distances with just one.

One easy way to continue is to take 0, 1, and everyone with distance 2, 3, or 4 from them. This is between 5 and 8 guests in total. We can apply the brute force from the previous subtask to find their order. Once we have this order, we can now easily extend it. We don't care about optimizing the number of questions yet, so we may simply ask one more question per guest. E.g., we process the remaining guests by increasing distance from $(0, 1)$. For each guest G, we look at the segment of guests we already have, take the two guests (X, Y) at one of its ends, and ask the query (X, Y, G). Answer 2 means that G is on this end, a bigger answer means she is on the opposite end.

One general direction of thought: partial solutions

A *partial solution* will be a sorted collection of guests for whom we already know their relative positions (that is, up to shifting all of them by the same amount and/or flipping their order). For example, the partial solution $(0, A)$, $(3, B)$, $(4, D)$, $(10, C)$ means that for some x in one full solution the guests A, B, C, and D are on seats $0 + x$, $3 + x$, $10 + x$, and $4 + x$.

Below we will see multiple ways to get a good solution. One of them will be based on finding efficient ways to do two things: to construct small partial solutions and to merge them.

Partial solutions via brute force

The brute force approach above can be generalized to an algorithm where we pick some specific small group of k guests and determine the partial solution for the chosen group. In places where the original algorithm used distance $N - 1$ the updated version will use the largest of all actually received answers. In places where the original algorithm used distance 2 the updated version will use the smallest answer that involves the fixed subset of guests.

For example, we can get a partial solution for any 5 guests in 10 questions.

Adding one guest to a partial solution

Given a partial solution and one new guest, we can easily add the guest to the partial solution in two questions with very little casework. Without loss of generality, let the partial solution consist of guests $0, \ldots, k - 1$ in this order and let k be the new guest. We can then always determine the correct position of guest k from the answers to the questions $(0, 1, k)$ and $(k - 1, k - 2, k)$.

This gives us our first full solution: start with 5 guests in 10 questions, and then repeatedly add another guest in 2 questions to that partial solution. After $2N = 4000$ questions we are done. This solution scored 65 points. Getting the rest will be more tricky.

Merging two partial solutions (slow-ish)

Another useful observation is that when merging two partial solutions into one, we only need a little extra information. In particular, in order to merge the two partial solutions it is sufficient to take any two guests from one partial solution and determine their positions within the other.

We can do that by applying the algorithm from the previous section twice. This gives us an easy way to merge any two partial solutions by asking four additional questions.

And that gives us another full solution: We can divide the N guests into $N/5$ groups of five, for each group determine their partial solution, and then perform $(N/5) - 1$ merges to produce one full solution. Using the approaches we already know we get that we need $10(N/5) = 2N$ questions in the first phase and then $4(N/5) - 4$ in the second, for a total of $2.8N - 4$ questions. For $N = 2000$ this solution needs 5596 questions. That's worse than the previous attempt, but there's a reason why we're mentioning this line of thought. Read on.

Doing the same thing more efficiently

By more careful casework it is always possible to construct the partial solution for 5 guests by asking just 6 out of the 10 possible questions. Merging can similarly be improved: there is a way to merge two partial solutions by asking just two questions instead of four.

Having just the first improvement brings the number of questions down to $2N - 4 \leq 3996$. Having just the second improvement gives us $2.4N - 2 \leq 4798$. If we find and implement both, we'll get a solution that needs just $1.6N - 2 \leq 3198$ questions.

Five guests in six queries

If we take four guests and ask all four possible questions about them, we are able to determine almost everything about them. There are always exactly two partial solutions consistent with the answers we got, and their only difference is that the identities of the middle two guests are swapped.

Without loss of generality, suppose that one of these two partial solutions has the four guests in order 0, 1, 2, 3 and let their relative coordinates, left-to-right, be $0, a, b, c$. (The other partial solution has the guests in the order 0, 2, 1, 3 at the same relative coordinates.)

We will now add a fifth guest (guest 4) and we will make two additional queries: $(0, 1, 4)$ and $(1, 3, 4)$. Below we'll show a constructive proof that these two queries will always determine the full partial solution for these five guests – that is, we'll be able to tell the order of guests 1 and 2, and also determine the coordinate where guest 4 belongs.

The new guest sits outside the range of the other four if and only if at least one of the two new distances is bigger than the distance between 0 and 3. When this happens, the bigger new distance tells us the exact position of guest 4, and then the smaller new distance tells us the coordinate of guest 1, which disambiguates between guests 1 and 2.

If we got here, we know that guest 4's coordinate is somewhere in the range $(0, c)$. Moreover, in this setting guest 4 will always be either between guests 0 and 1, or between guests 1 and 4. Hence, exactly one of the two new queries will return the answer matching the distance between 1 and the corresponding endpoint, and the other new query will return an answer greater than the distance between 1 and the other endpoint.

Each of the four cases is handled in the same way. For example, if the query $(0, 1, 4)$ returns the distance a, this is only possible if guest 1 is at a and guest 4 is between 0 and 1. If this happens, the other new query tells us the distance between guests 3 and 4 and we are done.

Note that it is not possible for both new queries to match some of the original distances. In particular, the two new queries cannot return distances b and $c - a$ because this would put guest 4 at the same position as one of guests 1 and 2.

Faster merging

We omit the details to make this writeup at least a bit shorter. There's nothing interesting, just more careful casework than in the four-query version. In one solution the first step is to take the two points at one end of the shorter partial solution and one point at an end of the longer partial solution. The second step is to show that a good second query can always be made afterwards.

Solutions based on endpoints

It is possible to solve the task using many different strategies. One type of strategies worth mentioning is that we can focus on identifying some guests who sit close to the edge of the row, and then use questions that involve those guests to find unique placement for everybody.

For example, start with all questions of the form $(0, 1, X)$, find the guest A that's farthest away (that one is surely on the end of the row), then among the others who are almost as far away from $0, 1$ as A find the one guest B who's A's neighbor (do you see how?), and then use queries (A, B, Y) to place everyone else.

(This solution has some additional special cases that need to be addressed. For example, what should we do when the distance between guests 0 and 1 is already $N - 1$ or $N - 2$?)

Solutions based on a pivot

Suppose we have two guests A and B such that we know their distance. We can use them to ask many queries of the form (A, B, x). When we do that, the pair (A, B) will be called a pivot. There are two basic types of answers we can get: either we learn that the answer to (A, B, x) is the same as distance AB, or that it's bigger. In the former case x lies inside the segment AB. In the latter case, we have two possible locations for x, symmetric around AB.

What happens if we have such a pair and we query all other guests? If the distance AB is d, we will get the set of $d-1$ guests that are inside the segment, then a pair of guests that are just to the left of A and just to the right of B, in some order (these are the two guests for whom the query returns $d+1$), then another pair, ..., and from some threshold distance there's only one guest per distance. (If the center of AB was the center of the whole set, we'll only see pairs, and the farther to the side AB is located, the fewer pairs and the more individual guests we'll see.)

The key observations here are:

- Each pair can be disambiguated using a single additional query: if we determine the location for one of the guests, the other must simply be in the other possible location they share.
- All individual guests can be resolved quickly together because it's enough to resolve the outermost guest – all others must be on the same side of AB so that there are no gaps in the seating order.

This approach will give us a new best solution as follows:

- Use any known approach to find the partial solution for roughly $\sqrt{N}$ guests.
- Pick the closest two among them as A and B = the pivot. Note that their distance is at most $\sqrt{N}$.
- Ask the queries with the pivot for all other guests. This will require $N - \sqrt{N}$ queries.
- Resolve guests that now have a fixed location (the other was occupied by a known guest).
- For each of the roughly $\sqrt{N}$ guests inside AB ask another query to find their location.
- For each of the (at most slightly fewer than $N/2$) pairs of guests around the pivot ask one question to learn who's where.
- Resolve the individual guests left over.

This solution requires $N + N/2 + O(\sqrt{N})$ queries.

For $N = 2000$ we can choose 50 for the first phase, and we'll get everything done in fewer than 3050 questions.

Another primitive operation: two pivots, two guests, three queries.

Suppose we already have two pivots (A, B) and (C, D) such that we know all their relative positions a, b, c, d and these satisfy $a < b < c < d$.

We can now take two new unknown guests E and F and make the queries (A, B, E) and (C, D, F).

For now, let's assume that both queries returned a value greater than the respective distance, that is, E lies outside the segment AB and F lies outside the segment CD. In other words, we have exactly two possible positions for E and two for F.

The important new observation is that in this setting we can always find the correct locations for both E and F using just a single additional question. To show this, we just need to show that we can always ask a question such that the four possible locations of the pair (E, F) will lead to four different answers.

Let the possible positions for E and F be $e_1 < e_2$ and $f_1 < f_2$. Clearly we have $e_1 < a < b < e_2$ and $f_1 < c < d < f_2$. We will show that we can always choose one of the four original guests as X and ask the query (E, F, X).

If the correct positions are e_1 and f_2, the answer will be $f_2 - e_1$ and it will be greater than the other three possible answers, regarless of who's X. We need to show that for at least one of the four possible X the other three distances will be distinct.

We'll omit the details and just note that the general idea is to consider each pair of distances separately and show that they can only be equal for at most one of the four points A, B, C, D.

In code we don't need the details of this proof. Instead, we can simply iterate over all four candidates and pick one for whom the three answers are distinct.

One possible solution based on two pivots

Start by finding the relative positions of five guests. Take first four of them as the two initial pivots A, B and C, D. Start making queries with the pivot + one new guest.

Whenever the guest falls inside the pivot's interval, make another query with the same guest + the other pivot to find this guest's location. The guest now divides the pivot into two intervals. Take the shorter of them as the new pivot, and stay with that pivot for the next query.

Whenever you have two queries that match the previous section, make a third query to localize that guest.

As we are at least halving the pivot's interval each time a guest falls inside, there will be at most $2 \log_2 N$ such guests. These require two queries each.

The remaining N' guests are placed using $3N'/2$ queries. This gives us another solution with roughly $3N/2 = 3000$ queries. Notably, this one is incremental: we can stop after any X and have a partial solution for X guests in roughly $3X/2$ queries.

$N = 40$ in guaranteed 54 queries – overview

In next section we will write a complete constructive proof that $N = 40$ is solvable in at most 54 queries. This is a somewhat optimized version of a general solution that can solve any input of size N in worst-case roughly $5N/4$ queries. There will be a lot of casework and a lot of technical details, so we'll start by summarizing the main ideas.

- Find the relative locations of 15 out of the 40 guests.
- Among those 15, pick two pivots. These will be pairs of adjacent or almost-adjacent guests.
- For each of the other 25 guests ask the query with this guest + one of the pivots.
- For each of those 25 guest we now have two possible positions.
- Use logic to resolve some of the cases that can occur without asking any queries.
- Realize that we can sometimes place not just two but four guests in a single query, as long as some specific conditions are met.
- Make such queries while you can and then resolve the rest by using a clever binary search.

That's it for the overview. If you want all the gory details, brace for impact and start reading on. You've been warned :)

Full writeup: $N = 40$ in guaranteed 54 queries

We'll start by using the two building blocks derived above: we can use 3×6 queries to solve three blocks of 5 elements, and then 2×2 queries to merge those partial solutions. We spent 22 queries and we have a partial order of 15 elements.

The sum of the 14 distances between adjacent elements is at most 39. We can now easily show that the two smallest among those distances must both be smaller than 3. This means that in each of these pairs we have either two adjacent guests, or two guests with just a single unknown guest between them.

Let's label the guests forming these two pairs (A, B) and (C, D) in order, with $A < B \leq C < D$. These will be our pivots.

In the second phase we will spend one query for each other element X, asking either the query (A, B, X) or the query (C, D, X). For each pivot, we can at most once get the special answer that X lies between the endpoints. If that happens, we know X's exact location, as the gap only had one empty space. In all other cases we will get two possible locations for X, symmetric around the center of the pivot used.

We will alternate between the two pivots in such a way that the final counts of non-special elements for each pivot differ by at most one.

We are at 47 queries asked, and the worst case that can happen is that both pivots were formed by two adjacent guests, and we now have 13 guests with two possible locations symmetric around AB and 12 with locations symmetric around CD.

Some of these guests will come in pairs that have the same distance from the same pivot. Each such pair has exactly two possible configurations, and also enforces that no other element can be in either of those locations. We like these, because they are basically just two additional known positions. We'll treat their positions as known and we'll only resolve them at the very end.

Note that at this point unresolved guests who share the same pivot have mutually disjoint pairs of possible positions.

At this point we can apply several greedy rules to resolve stuff without asking any queries:

- All positions between the smallest and the largest currently known position must be occupied. If there is exactly one guest that can be there, place them there.
- We can maintain an interval of positions that *may* be occupied. We can initialize it to the interval that starts at max(known positions)−39 and ends at min(known positions)+39.
- Whenever we see a position within that interval that cannot be occupied, we can shrink the interval.
- Whenever we have a guest whose one possible position lies outside the possible interval, we place them at the other position.

Once none of these greedy rules apply, we are left with a contiguous range of positions, each with either a known guest or with at least one guest who *can* be there.

Let's model the situation as a graph. The integer coordinates that can still be occupied by someone will be the vertices and each unresolved guest will be an edge between their two possible vertices. We will color the edges red if they used the first pivot and blue if they used the second pivot.

As the remaining guests who share the same pivot don't share the same locations, each vertex has at most one incident edge of each color. Thus, each connected component must be either an alternating path (possibly consisting of just one edge) or an alternating cycle.

Alternating cycles are impossible. To note why, first observe that the two pivots have distinct centers and therefore distinct sums of coordinates. For any guest, the sum of coordinates of their two options matches the sum of their pivot. If we had an alternating cycle, we could count as follows: Look at just the k red edges. Together, they cover all vertices on the cycle. For each of them the sum of its

two vertices is the sum of the first pivot, thus the total sum of all coordinates is k times the sum of the first pivot. Of course, we can also look at just the k blue edges and this produces the contradiction we seek.

Thus, each component is an alternating path. Each of the x paths has two ends: locations where just that single guest can sit. In total, we now have $2x$ such vertices. On each path exactly one vertex will remain unused. Thus, our graph (counting also all vertices that are known to have a guest) must currently have exactly $40 + x$ vertices. Out of those, some contiguous segment of length 40 is actually occupied.

From each alternating path, the one unused vertex only has two options: it can only be either the leftmost or the rightmost of its vertices – otherwise we would have a gap somewhere inside the segment of used vertices.

At this point we can add a new greedy rule: if there is an alternating path with one end inside the segment of the currently known positions, that end must be occupied, and thus its opposite end isn't, and we can resolve it without any queries.

Once none of the greedy rules including the new one apply, we know that each unresolved alternating path has its leftmost vertex to the left and its rightmost vertex to the right of all known positions.

Additionally, one of the ends (i.e., degree-1 vertices) of the alternating path will always be on its outside (i.e., it will be either the leftmost or the rightmost of its vertices). Thus, we can resolve the entire path by resolving the guest who can sit at this vertex. If the guest is there, the other extremal vertex must be empty and vice versa. (There's a cute proof similar to the one for cycles. Note that if we take any point, mirror it around one pivot and then around the other pivot, we have just moved it by twice the distance of their centers. Thus, if we number the vertices along the path, the coordinates of all odd-numbered vertices will form one arithmetic progression and the coordinates of all even-numbered vertices will form another. Our claim then follows easily.)

Now we are ready to resolve everything. The only remaining question is: what is the smallest number of questions in which it's always doable?

We can easily resolve one guest using one question. However, we can also resolve a red guest and a blue guest (i.e., two guests who were queried with different pivots) together using just a single query: we can always pick one of the guests with a known position such that the four possible answers all give different answers. (This is a non-trivial statement and we are omitting the proof intentionally because this text is already way too long.)

There are two things to resolve: if there are pairs of guests who have the same two possible locations, we need to tell which is which, and we need to find which 40 contiguous positions are actually used – i.e., for each path, which one of its endpoints is unused.

For each path, label its left endpoint dark blue if it can be resolved by querying a blue guest, or dark red otherwise. We can precisely determine the set of used positions by finding the leftmost used dark blue and the leftmost used dark red vertex. Each of these questions can be resolved by binary search. E.g., if we have seven dark red locations, we just need to query three red guests.

We will do two processes in parallel. One process is that we want to resolve all blue pairs and then binary search the dark blue positions, the other process is the same in red. In each step we take the next blue guest from one process, the next red guest from the other process and spend one query to resolve both of them together.

We can easily verify (e.g., by not being smart and just looking at all possible cases) that regardless of how many blue and red pairs there are, in the worst case (which occurs when all unpaired edges are length-1 alternating paths that still survived) we will only need 7 queries to resolve everything. Thus we are done after at most 47+7 = 54 queries, q.e.d.

General solution in roughly 5N/4 queries

This approach generalizes to an ($5N/4$ + something small) solution. If we don't care about the "something small" as much as for $N = 40$, it gets somewhat easier to implement.

One possible implementation along these lines looks as follows: - Use a slower solution that incrementally constructs a partial solution for roughly the first $\sqrt{N}$ guests. - In the obtained partial solution, pick two adjacent pairs of guests with smallest distances as the two pivots. - Proceed as above, with the small differenence that now there can be $O(\sqrt{N})$ moments when the queried guest falls into the pivot's interval. In each of these we simply ask one more question with that guest and the other pivot to place them, and then we continue with the same pivot and the next unknown guest.

We think we know how to improve the 5/4 constant at least a bit further, but there's not enough room in this margin for that proof :)

Lower bound

In this section we'll look at the problem from the opposite side: can we guarantee that all solutions *must* ask at least some number of questions in order to be correct? Sure we can. An easy lower bound is around $N/3$ queries: if there are multiple guests that haven't been mentioned in any query, there are clearly multiple possible permutations (the unknown guests can be swapped arbitrarily) and therefore the solution cannot yet be sure that it knows the answer. Below we'll show a significantly stronger lower bound.

When solving a test case we need to determine which one out of the $N!/2$ possible seating plans is the correct one. To do that, we need to obtain $\log_2(N!/2)$ bits of information, which is approximately $N \log_2 N - 1.44N$ bits of information.

Each query returns one of only $N-2$ possible answers, so it can give us at most $\log_2(N-2)$ bits of useful information.

The ratio between those two amounts tells us a lower bound on the number of questions any solution needs to make in the worst case.

If you aren't comfortable working with information, the same argument can be phrased as a purely counting argument. E.g., assume that you have a deterministic solution that can correctly solve any input of size N and always makes at most k queries. Now imagine all possible executions of our solution on inputs of size N as a huge decision tree: the program runs, then it outputs the first query and reads the answer, then based on that answer it computes the second query it wants to ask, and so on. Leaves of this tree correspond to moments when the program gives its answer.

As each of the queries has at most $N-2$ possible answers, there are at most $(N-2)^k$ different ways in which the program can run, i.e., at most $(N-2)^k$ leaves in the decision tree we just described. In other words, the program can give at most $(N-2)^k$ different outputs.

On the other hand, we know that there are $N!/2$ possible answers, and a correct program must be able to give each of those. Thus, the number of leaves in the decision tree must be at least as big as the number of possible answers.

And therefore for any deterministic solution the number k of queries asked in the worst case must be big enough for the inequality $(N-2)^k \geq N!/2$ to hold.

This estimate tells us that we need at least $k = 31$ questions for $N = 40$, and at least $k = 1738$ questions for $N = 2000$.

Of course, this is not the true optimal number of questions, the lower bound is not tight. Such a low number of questions would require each question to basically exactly split the space of remaining valid answers into $N-2$ equally large buckets (just like binary search always splits the search space into two equal halves), and a more careful analysis can show that in our problem this is not possible – especially once we already have some partial information, it will be the case that for every possible query some answers will be completely and others almost completely impossible. Thus, each actual query we can make will give us fewer actual bits of information (in the worst case) than what we need for the theoretical lower bound.

Optimality?

So, where's the boundary? What is the optimal solution? Is there a solution that always needs at most N queries? Or even one that can just needs $0.99N$ queries? We don't know. Maybe you can tell us if you find one or disprove it by showing a stronger lower bound :)

We had multiple solutions where we could prove that in the *expected* case the number of questions they need is N + just a very small function, e.g.,

$N + O(\log N)$. This, however, is one of the problems where the expected case is much easier than the worst case.

The best solutions the SC had did solve all the prepared test cases and antagonistic strategies for $N = 40$ using at most 40 queries, but we don't have a proof that their worst case isn't higher. One such solution was using a heavy-duty solver (or-tools) to sample the space of valid solutions and ask queries that made sense for them. Another, slightly worse but self-contained solution worked as follows: We split the guests into four groups of ten, asked some (usually 7-8) questions within each group until the number of possible partial solutions became small enough, then combined those partial solutions to get a set of permutations that are consistent with all four groups (merge two, merge the other two, and then iterate over one set of partial solutions for 20 guests and look for essentially its complement in the other set of partial solutions). Finally, we greedily asked questions to eliminate all wrong permutations, always choosing the question that leaves the fewest candidates in the worst case.