46th ICPC World Finals

Solution sketches

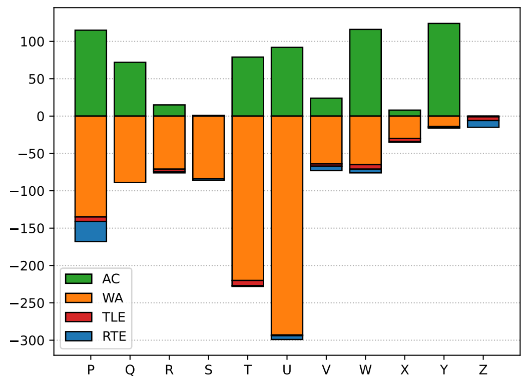
Disclaimer *This is an unofficial analysis of some possible ways to solve the problems of the 46th ICPC World Finals. They are not intended to give a complete solution, but rather to outline some approach that can be used to solve the problem. If some of the terminology or algorithms mentioned below are not familiar to you, your favorite search engine should be able to help. If you find an error, please send an e-mail to* `per.austrin@gmail.com` *about it.*

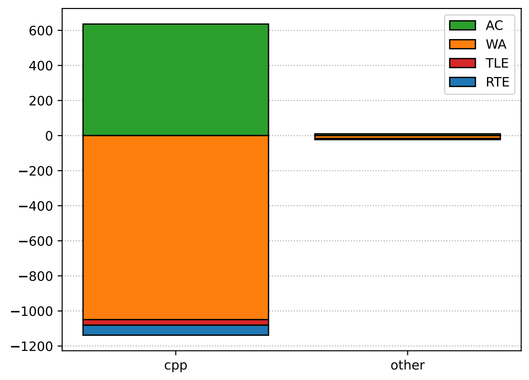
— Per Austrin, Arnav Sastry, Paul Wild, and Jakub Onufry Wojtaszczyk

Summary

Congratulations to Peking University
for the title as the 46th ICPC World Champions!

As general statistics, here are two graphs showing the number of submissions made for each problem and for each programming language, respectively. The positive y axis has the number of accepted solutions made, and the negative y axis has the number of rejected solutions made.





C++ is by far the most dominant language. Across the 46th and 47th World Finals (which were held at the same time, about 1% of submissions were made in Python, 0.5% of submissions in Kotlin, and the remaining 98.5% of submissions were in C++. There were no Java submissions.

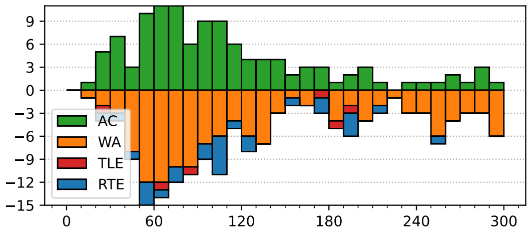
A note about solution sizes: below the size of the smallest judge and team solutions for each problem are given. These numbers are just there to give an indication of the order of magnitude. The judge solutions were not written to be minimal (though some of us may overcompactify our code) and can trivially be made shorter by removing spacing, renaming variables, and so on. And of course the same goes for the code written by the teams during the contest!

Explanation of activity graphs: below, for each problem a team activity graph is shown. This graph shows the number of submissions of different types made over the course of the contest: the x axis is the time in minutes, the positive y axis has the number of accepted solutions made, and the negative y axis has the number of rejected solutions made.

Problem P: Turning Red

Problem authors:
Jakub Onufry Wojtaszczyk and Arnav Sastry

Solved by 115 teams.
First solved after 19 minutes.
Shortest team solution: 1691 bytes.
Shortest judge solution: 1341 bytes.



This was a relatively easy problem. Let us model the three colors as red $= 0$, green $= 1$, and blue $= 2$. Then the effect of pressing a button is that all the lights controlled by the button are incremented by 1 modulo 3.

Let x_i be the (unknown) number of times we press button i. Then the requirement that a light ℓ controlled by buttons i and j must turn red becomes the equation $c_\ell + x_i + x_j = 0 \pmod 3$, where c_ℓ denotes the initial color of this light. Note that if either x_i or x_j is known, then this equation lets us immediately calculate the value of the other one. This means that if we consider the implied graph where two buttons i and j are connected if they control the same light, then based on the value x_i for one button we can propagate and calculate the values within the entire connected component of that button.

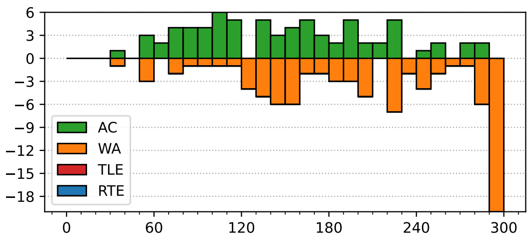
This leads to the following linear-time algorithm: for each connected component of the graph, pick an arbitrary button i, try all 3 possible values $x_i = 0, 1, 2$, and propagate the result. If an inconsistency is found (some equation is not satisfied), then this value of x_i is invalid. Otherwise, check the total number of button presses (sum of x_i's in the component). If all 3 possible choices of x_i are invalid then there is no solution, otherwise pick the one that leads to the smallest number of button presses within this component.

In the problem there could also be some lights that were controlled by a single button, leading to an equation of the form $c_\ell + x_i = 0 \pmod 3$ which immediately determines x_i. One could special-case the handling of these and propagate their values first, but it is probably less error-prone to simply include them in the inconsistency check in the above algorithm instead.

Problem Q: Doing the Container Shuffle

Problem authors:
Matthias Ruhl and Derek Kisman

Solved by 72 teams.
First solved after 32 minutes.
Shortest team solution: 513 bytes.
Shortest judge solution: 407 bytes.



Define the "joint order" on the containers that are still on the two stacks as follows. Assume $a_1, \ldots, a_k$ are the containers on the first stack, from bottom to top, and $b_1, \ldots, b_l$ are the containers on the second stack from bottom to top. Then the "joint order" is $a_1, \ldots, a_k, b_l, \ldots, b_1$.

Take any two containers i and j. Notice that at any point in the unloading process, before either i or j are loaded onto a truck, the joint-order interval between i and j contains the containers that were in the interval between i and j before the loading started, minus the containers that were already loaded onto the trucks.

Let us now say we have two containers a_i and a_{i+1}, and we want to know how many containers we will need to move before unloading a_{i+1}. The answer is exactly the set of containers in the joint-order interval a_i, a_{i+1}.

So, now we need to figure out the expected number of containers that are in the joint-order interval a_i, a_{i+1} at the time when we try to unload a_{i+1}. We want to check for some container v what is the probability that v is in this interval.

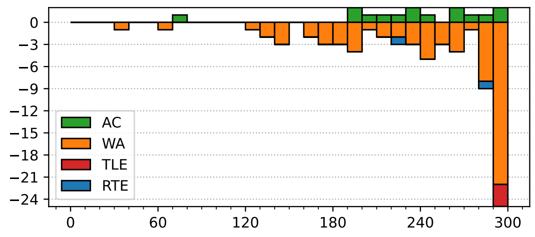
If v got unloaded before a_i, the probability is zero. Otherwise, the probability is equal to the probability that v is in the joint-order interval at the beginning of the loading. And this probability is 0 if $v < \min(a_i, a_{i+1})$ in the unloading order, and $1/2$ if $v > \min(a_i, a_{i+1})$ in the unloading order.

Actually calculating the number of containers smaller than $\min(a_i, a_{i+1})$ that are also loaded after a_i can be done in $O(\log n)$ for a single container using a data structure that supports point updates and range sums, such as a range tree or a Fenwick tree.

Problem R: Zoo Management

Problem authors:
Jakub Onufry Wojtaszczyk and Derek Kisman

Solved by 15 teams.
First solved after 76 minutes.
Shortest team solution: 3475 bytes.
Shortest judge solution: 3163 bytes.



First, we may observe that all bridges can be removed from the graph without changing the result, as no cycle passes through them. This leaves us with several independent subproblems, one for each 2-edge-connected component.

Now we consider a component where every edge is a part of a cycle. If the component is a simple cycle, then the problem becomes checking cyclic equivalence of strings, which can be done using any linear-time string matching algorithm.

If the component has more than one cycle, then the answer is either "any permutation can be achieved" or "any even permutation can be achieved". We omit most of the proof details here, but a general idea for proving such claims is to first find a way of making a single swap of any two labels, and then show that one can combine this swap with the given cycles to make arbitrary swaps, and thus arbitrary permutations. Similarly, for the even permutations, one first finds a way to make a single 3-cycle, then arbitrary 3-cycles, then arbitrary even permutations.

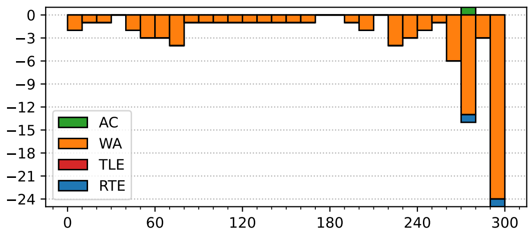
First, consider the case where there's an edge that belongs to more than one cycle, or equivalently, where there are two vertices connected by three edge-disjoint paths. By rotating along the three cycles that can be formed by choosing any two of these paths, one can achieve any permutation of the labels in this subgraph, and one can also show that this allows achieving any permutation on the entire component.

If the component consists of multiple cycles which only share single vertices, then we can take two adjacent cycles, say a and b, and perform the permutation $aba^{-1}b^{-1}$, which results in a cycle of three of the labels. By the general strategy laid out above we can therefore make all even permutations. If at least one of the cycles has even length (and thus is an odd permutation), then it turns out we can also make all odd permutations. Otherwise, only even permutations are achievable.

Problem S: Bridging the Gap

Problem authors:
Walter Guttmann and Paul Wild

Solved by 1 team.
First solved after 276 minutes.
Shortest team solution: 1433 bytes.
Shortest judge solution: 857 bytes.



There are a number of observations that can be made about the structure of the solution:

- Every time a group crosses backwards, the group is only one person
- Every time a group crosses forwards, the group is either:
  - the k slowest people and $c - k$ fastest people who will later cross back, or
  - some k of the fastest people, who will later cross back, or
  - all of the people left, assuming there are no more than c of them.

After every forward crossing except the last, we need to perform one back-crossing. So, we can instead pay for the back-crossing when we cross the fast walkers forward, and keep track of how many paid-for back-crossings we have accumulated.

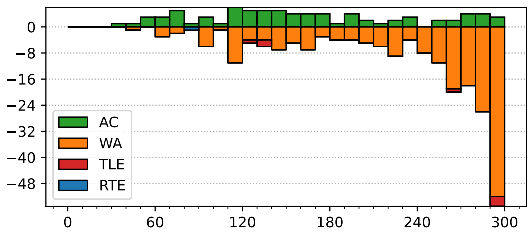
This naturally translates to a dynamic programming solution, where we calculate the cost of having the k slowest people already crossed, and l back-crossings earned. Naively, this is $O(n^3)$ — we have n^2 states, and up to n transitions out of each state. But, if you look closer, it never makes sense to accumulate more that n/c backcrossings (since that's already enough to cross everyone over), so the state space is $O(n^2/c)$. At the same time, the number of transitions out of a state is $O(c)$, so the actual cost after filtering out unreachable states and nonexistent transitions is $O(n^2)$.

While the resulting code is short, a number of judges found the implementation to be tricky to get right.

Problem T: Carl's Vacation

Problem authors:
Arnav Sastry and the World Finals judges

Solved by 79 teams.
First solved after 33 minutes.
Shortest team solution: 1805 bytes.
Shortest judge solution: 893 bytes.



Clearly, the challenge in this problem is not about running time, but rather about how to represent the three-dimensional geometry problem in a convenient way.

On a plane, the shortest path between two points is always a line segment. So, the path will be a segment from the top of one pyramid to its base, then a segment across the ground to the base of the other pyramid, and then a segment from the base to the top of that pyramid.

We have 16 ways of selecting the faces of the two pyramids we will travel on, so we will check all of them. Now, we have two tilted triangles, each attached to the ground by the base, and we need to get from the top of one triangle to the top of the other using as short of a path as possible. The answer will be the same if we rotate each triangle around its base and flatten it out — so we now just need to find the shortest path between two points on the plane, under the condition that it passes through two specified segments. The shortest path between two points on a plane is just a segment. So, to summarize:

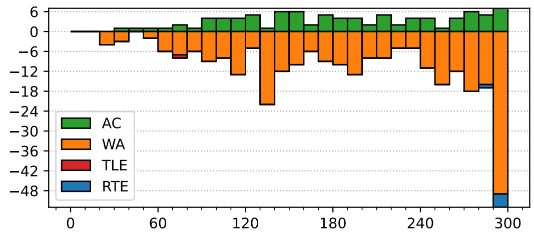
- We iterate over all selections of the faces on both pyramids.
- We rotate the top of the pyramid around the line containing the base of the selected face so that the top of the pyramid lands on the ground.
- We check if the segment between the two rotated tops intersects the two bases, in the right order.
- If yes, we take the distance between the rotated tops as a candidate distance.
- We output the smallest candidate distance.

Another approach is to observe that, for a pair of pyramid faces, we can parameterize the path taken by the ant by two angles θ_1 and θ_2. The minimum of $dist(\theta_1, \theta_2)$ can be found using golden section or ternary search.

Problem U: Toy Train Tracks

Problem authors:
Matthias Ruhl and the World Finals judges

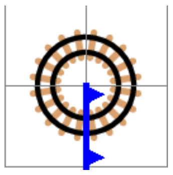
Solved by 92 teams.
First solved after 31 minutes.
Shortest team solution: 668 bytes.
Shortest judge solution: 464 bytes.

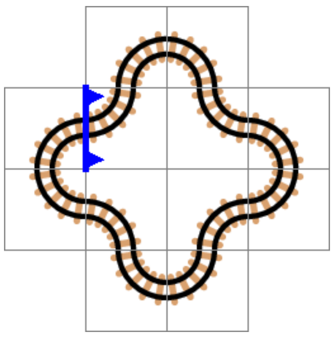


Constructive problems such as these have many different approaches. Here is one approach by one judge. Any closed loop must have an even number of curved track segments and straight track segments, so we can start by making both even. There are then three basic cases to handle:

1. there are no straight track segments,
2. $c \equiv 0 \pmod 4$, or
3. $c \equiv 2 \pmod 4$.

If there are no straight track segments, then we can make one of two "base" tracks shown in Figure 1. You can then use repeated iterations of "`RL`" on opposite sides of Figure 1b to extend the track by 4 curves at a time, as shown in Figure 1c. Note that there is no way to make a track with exactly 8 curved segments.


(a) Base track: `LLLL`


(b) Base track: `LRRLRRLRRLRR`

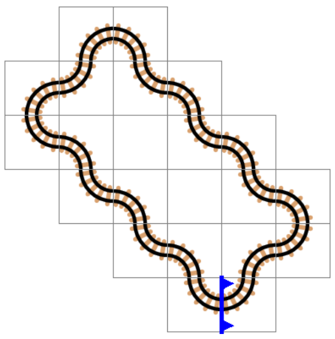
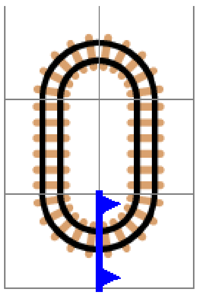

(c) `LRLLRLRLRLLRLLRLRLRL`

Figure 1: The all-curves case.

On the other hand, if there are at least 2 straight track segments, we then branch on the remainder of $c \pmod 4$. Because we have forced c and s to be even, this is either 0 or 2. Two possible "base" tracks are then shown in Figure 2, and we can extend these figures with 2 straight track segments placed on opposite sides, as well as with the "RL" extensions described above.


(a) LSLLSL

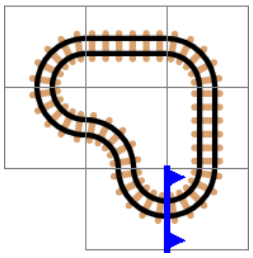
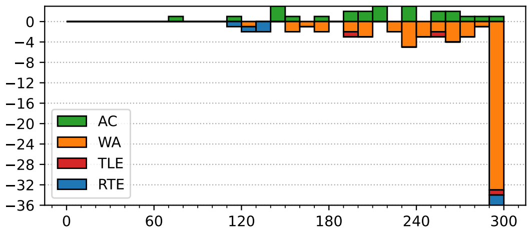

(b) LSLSLLRL

Figure 2: Two base tracks for the case with straight track segments.

Problem V: Three Kinds of Dice

Problem authors:
Derek Kisman and Walter Guttmann

Solved by 24 teams.
First solved after 76 minutes.
Shortest team solution: 1545 bytes.
Shortest judge solution: 871 bytes.



Consider a face of die D_3, with value v. Let $S_i(v)$, for $i \in \{1,2\}$, be the expected value of points die D_i gets if D_3 lands on this face. The expected value die D_1 gets overall is the average of $S_1(v)$ over all faces of D_3.

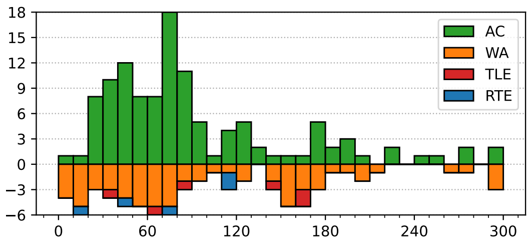
$S_1(v)$ is simply the number of faces of D_1 larger than v, plus half the number of faces equal to v. The same goes for $S_2(v)$. This means that there are $O(n)$ possible values for the pair $(S_1(v), S_2(v))$, and we can easily calculate all of them in $O(n \log n)$. We are now looking to assign weights to those points so that the weighted average of $S_1(v)$ is ≤ 0.5, and the weighted average of $S_2(v)$ is as high as possible (and vice versa).

This is a geometry problem. Note that all the points we can obtain by weighted averages of a set of points is exactly their convex hull; so we're looking for the highest y-variable value in the convex hull intersected with $x \leq 0.5$, which can be determined in $O(n)$ in a number of ways.

Problem W: Riddle Of The Sphinx

Problem authors:
Martin Kacer and Per Austrin

Solved by 116 teams.
First solved after 8 minutes.
Shortest team solution: 478 bytes.
Shortest judge solution: 347 bytes.



This was an easy interactive problem and there are many ways to solve it. A general observation for this type of problem (which is maybe a bit overkill for the present situation) is the following. Consider the 5×3 matrix

$$A = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \\ a_{41} & a_{42} & a_{43} \\ a_{51} & a_{52} & a_{53} \end{pmatrix}$$

where the i'th row are the three numbers you give in your i'th question. Then, if any 3 rows of A are linearly independent, we can uniquely determine the correct answer. This is the case because if we remove one of the truthful answers, we will have an inconsistent system of equation, but if we remove the lie, then we will have a consistent overdetermined system of equations. In other words we can uniquely identify which answer is the lie, and then we can recover the correct answer using any three of the other answers.

Since we can choose A freely it is nicest to choose it in such a way that the answer is easy to recover, e.g.

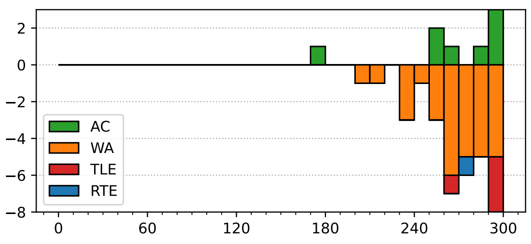
$$A = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 1 & 1 \\ 1 & 2 & 3 \end{pmatrix}$$

Note that we cannot change the last query to $(1, 1, 2)$, because then the last three answers would be linearly dependent, and if one of he first two questions was a lie we would not be able to figure out which one of them was a lie. Similarly, $(0, 1, 2)$ does not work as the last query, as then the second, third and last question are linearly dependent.

Problem X: Quartets

Problem authors:
Martin Kacer and Yujie An

Solved by 8 teams.
First solved after 179 minutes.
Shortest team solution: 3096 bytes.
Shortest judge solution: 2894 bytes.



To solve this problem, we need to understand what data a single move provides us about the state of the world and determine whether the information we have is compatible with some starting positions of the cards. Given that we have to check consistency of our knowledge with possible starting hands, we will represent our knowledge in terms of what it implies about the starting hands (and separately keep how that state changed from the time we started).

There are three types of events listed in the input, we need to understand how they affect the state of the world. We will store the following types of information:

- A specific card started in the hand of player P
- A specific card did not start in the hand of player P
- A specific player started with some card in set S, which has not moved yet

Additionally, for each card, we store its current position, if it has moved.

If a player x asks y for a card C, and gets it, then:

- If we don't know for sure that x has a card in the suit (we could know that because x is known to have a card in that suit now, or they're known to have an unknown card in that suit), we now mark x as having started with a card in the suit that we don't know yet.
- If we knew the current position of C, it was either in y's hand, in which case we learn nothing, or it was in someone else's hand, which means someone cheated.
- If we didn't know the current position of C, then we learn that it started in y's hand, and is currently in x's hand. Also, if we knew that y started with some unknown card of the suit, we now clear this information.

If a player x asks a player y for C and doesn't get it, then:

- We learn the same thing as above about x having a card in the suit,
- If we don't know the current position of C (which means it didn't move from the beginning), we know y didn't start with C.

Finally, if a player declares a quartet, we need to know that for all the cards in the set, either it has moved and they already have it (in which case we know nothing), or it hasn't moved, and then we learn they started with it. And we mark all the cards to have a known position of "gone".

Given this knowledge, how do we check if the information we have is consistent with some starting hand? The constraints are small enough that a somewhat pruned brute force search can work. However, the problem can also be represented as a matching problem, where we are matching the 32 cards in the players' hands to the 32 cards values.

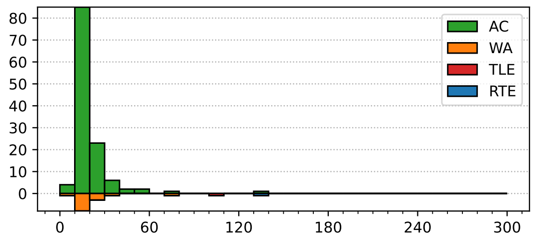
- If we know a specific card started in a player's hand, we just pick one of the eight cards in that player's hand, and hard-match it to the card values.
- If we know a specific card has not started in a player's hand, we remove the edges connecting every card in the player's hand to that card value.
- If we know a player has to start with a card from a set that hasn't moved, we pick one card in that player's hand, and remove all the edges connecting it to any other sets.

If there exists a perfect matching in this graph, it's possible no-one has cheated yet.

Problem Y: Compression

Problem authors:
Jakub Onufry Wojtaszczyk and Bob Roos

Solved by 124 teams.
First solved after 6 minutes.
Shortest team solution: 305 bytes.
Shortest judge solution: 108 bytes.



This was inteded to be one of the easiest problems of the contest. The three key properties to notice are that

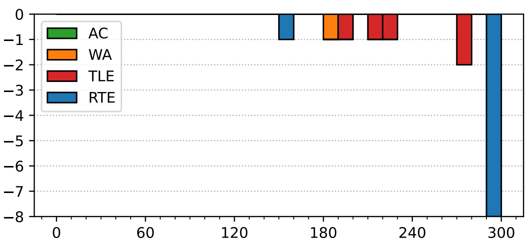
1. the first character of the string cannot change,
2. the last character of the string cannot change, and
3. it is impossible to erase all occurences of a character.

With this, the set of possible answers for a binary string are narrowed down to strings of length at most 3. This can be achieved by removing characters so all adjacent characters are different, and then removing prefixes of size 2. There is a unique shortest result for all binary strings.

Problem Z: Archeological Recovery

Problem authors:
Federico Glaudo and Per Austrin

Solved by 0 teams.
Shortest judge solution: 2537 bytes.



This was, in the judges' opinion, the hardest problem in this set.

Each pyramid position corresponds to a vector in $\mathbb{Z}_3$, with the ankh corresponding to 0, the eye to 1, and the ibis to 2. Similarly, each lever corresponds to a vector in $\mathbb{Z}_3^k$, with not moving a pyramid corresponding to 0, moving it clockwise corresponding to 1, and counterclockwise — to 2. Moving a lever corresponding to a vector u corresponds to adding the vector representing the lever to the vector representing the pyramids.

So, in the backstory, we had $n \leq 40$ lever vectors in $\mathbb{Z}_3^k$. We took all possible 2^n sums of some of those vectors, and in each case got some vector describing the positions of all the pyramids; call those vectors $(s_i)_{i=1}^{2^n}$. We are asked to find some possible sequence of vectors $(u_j)_{j=1}^n$ that would generate the observed s_i. This is a hard problem, and we will solve it in parts.

Part 1: $k = 1$ First, let us make an observation about the problem in dimension 1. In this case, (u_j) is just a multiset of numbers in $\mathbb{Z}_3$, and similarly (s_i) — so (s_i) is just a zeroes, b ones, and c twos. Let's see what are the effects on the multiset (s_i) of appending a single number to (u_j).

If we append a zero, then a, b, and c get multiplied by two. If we append a 1, then $a, b, c := a + c, b + a, c + b$ (and the results of appending a 2 are similar). If we first append all the 1s and 2s, we can see that the multiplicities (a, b, c) are always either $\{x, x+1, x+1\}$ or $\{x, x, x+1\}$, in some order. The important part is that at least one of the numbers a, b, c is always odd if we only add levers corresponding to one and two. So, the number of levers corresponding to zero is the largest power of two dividing all of a, b, c.

Part 2: Summing over a linear subspace Now, let's try to use the reasoning for $k = 1$ in the more general case. Consider any vector $x \in \mathbb{Z}_3^k$, and take the scalar product of everything with that vector. More precisely, consider the set of numbers $(u_j \cdot x)_{j=1}^n$; then the multiset of all possible sums of those numbers is $(s_i \cdot x)_{i=1}^{2^n}$, because scalar products commute with addition. So, from Part 1, we can find out how many u_js satisfy $u_j \cdot x = 0$, for any x. Let us denote $f(x) = |\{j : u_j \cdot x = 0\}$. There are only 3^k possible xs, and up to 3^k possible values of s_i, so we can calculate $f(x)$ for all possible x in $O(k3^{2k})$, which is easily fast enough.

This means that for any linear subspace of dimension $k - 1$, we can find out how many of the u_js are in that subspace. We can use this in a number of ways to narrow it down. For example, consider an arbitrary-dimension subspace H, and $G = H^\perp$, and look at the $\sum_{x\in H} f(x)$. This is

$$\sum_{x\in H} f(x) = \sum_{x\in H}\sum_{j=1}^{n}[x \perp u_j] = \sum_{j=1}^{n}\sum_{x\in H}[x \perp u_j] = \sum_{j=1}^{n} |\{H \cap u_j^\perp\}|,$$

which is the size of the intersection of H and $u_j^\perp$, summed over j. How much does a single u_j contribute to that sum? If $u_j \in G$, then u_j is orthogonal to every vector in H, and so it contributes $|H|$. Otherwise, the intersection of H and $u_j^\perp$ will be a subspace of dimension one smaller, and so u_j will contribute $|H|/3$. So, just by counting, for any G we can calculate how many u_js are in G — it is

$$\frac{1}{2}\left(\frac{3\sum_{x\in G^\perp} f(x)}{|G^\perp|} - n\right).$$

Having previously calculated $f(x)$ for any x, we can calculate the value above in $O(|G^\perp|)$ for any G.

Part 3: Flipping signs First, take $G = \{0\}$ — the zero-dimensional subspace. We find out how many all-zero levers are in the set $\{u_j\}$. Then, take $G = \{0, x, -x\}$ for any x. We find how many of the u_js are in G — and, since we already know how many u_js are zero, how many are in the set $\{x, -x\}$ for any $x \in \mathbb{Z}_3^k$. Calculating this for all $(3^k - 1)/2$ values of x takes $O(3^{2k})$. So, we know all the u_j up to the selection of signs; we now need one last idea.

We potentially have up to 2^n possibilities to assign the signs to check (possibly less, if we had any zero vectors, or if any group of $\{x, -x\}$ is larger than one element). We need to find one that works. Notice that if we calculate the sums s_i for some $u_1, \ldots, u_n$, then the sums s'_i for $u_1, \ldots, u_{n-1}, -u_n$ are going to be the multiset $\{s_i - u_n\}$ (the bijection is that we map any set of indices containing n to the same set without n, and vice versa). So, we do the following:

1. We pick any set of signs for u_i, and calculate the multiset s_i.
2. While doing this, for any element of the multiset s_i also remember any one way of reaching it as a sum of some u_is.
3. For any vector v in $\mathbb{Z}_3^k$, look at the multiset $\{s_i - v\}$, and check if it is equal to the multiset given on the input.
4. For any v that worked, see if it can be represented as a sum of some u_is, and how (this is what we stored the ways of reaching elements for in point 2).
5. If we do find one, then we flip the signs of the u_is that are used to reach v, leave the rest alone, and output that as a solution.
6. If none of the vs that lead to the input multiset can be represented as a sum of some u_is, output impossible.

You can also see this paper or this paper for a detailed examination of a more generic problem.