47th ICPC World Finals

Solution sketches

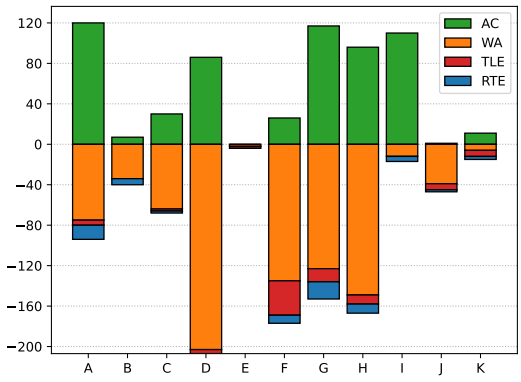
Disclaimer *This is an unofficial analysis of some possible ways to solve the problems of the 47th ICPC World Finals. They are not intended to give a complete solution, but rather to outline some approach that can be used to solve the problem. If some of the terminology or algorithms mentioned below 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.*

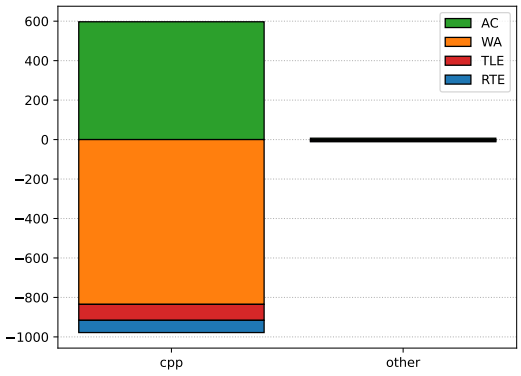
— Per Austrin, Arnav Sastry, Paul Wild, and Jakub Onufry Wojtaszczyk

Summary

Congratulations to National Research University Higher School of Economics
for the title as the 47th 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.

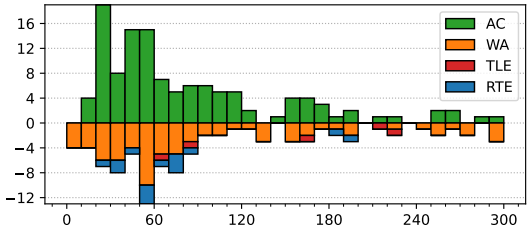
A note about solution sizes: below the size of the smallest judge and team solutions for each problem are given. These numbers are just there to give an indication of the order of magnitude. The judge solutions were not written to be minimal (though some of us may overcompactify our code) and can trivially 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 A: Riddle Of The Sphinx

Problem authors:
Martin Kacer and Per Austrin

Solved by 120 teams.
First solved after 10 minutes.
Shortest team solution: 532 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.

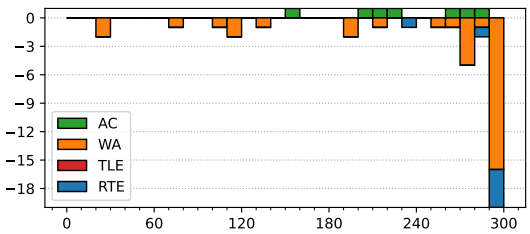
$$A = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 1 & 1 \\ 1 & 2 & 3 \end{pmatrix}$$

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 B: Schedule

Problem authors:
The World Finals judges and Bob Roos

Solved by 7 teams.
First solved after 153 minutes.
Shortest team solution: 1059 bytes.
Shortest judge solution: 532 bytes.



Suppose we have some schedule with isolation k. Then during the first k weeks, every pair of individuals on different teams meet at least once, and if we repeat the schedule for these first k weeks indefinitely, we get a periodic schedule for arbitrarily many weeks with the same

separation. This means that the problem is equivalent to finding the smallest k such that all people can meet at least once in k weeks. If $k \leq w$ then we take that schedule and repeat it to get a full schedule of w weeks, and if $k > w$ then the answer is `infinity`.

We can view a k-day schedule for one team as a binary string x of length k, with $x_i = 0$ indicating that the first team member comes to work on day i, and $x_i = 1$ indicating that the second team member comes to work on day i. Two binary strings x and y are compatible if for all four combinations $c \in \{00, 01, 10, 11\}$ there is some i such that $x_i y_i = c$. A schedule for n teams is then a list of n binary strings that are pairwise compatible.

We claim such a schedule exists if $n \leq \binom{k-1}{\lceil k/2 \rceil}$. To see this, consider enumerating all binary strings of length k with a 0 in the first bit, followed by some binary string of length $k-1$ with $\lceil k/2 \rceil$ ones. It is not hard to see that any pair of such strings are pairwise compatible: since they both have the same number of ones and are different strings, there must be some index where we see the combinations 01 and 10; since they both start with 0 there is an index where we see the combination 00; and since they both have more than $(k-1)/2$ ones among the last $k-1$ bits, there must be some index where we see the combination 11.

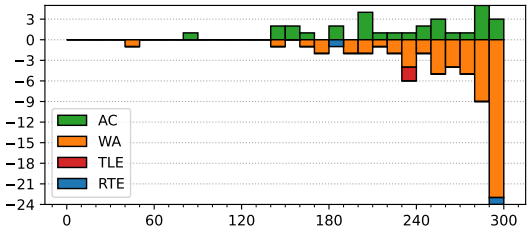
We also claim that such a schedule cannot exist if $n > \binom{k-1}{\lceil k/2 \rceil}$. To see this, note first that without loss of generality we can assume that all strings in a schedule start with 0 (because if we flip 0 and 1 in a string, it remains compatible with the same strings). Also, since there must both be some i such that $x_i y_i = 01$ and some j such that $x_j y_j = 10$, any list of pairwise compatible schedules must be an *antichain* in the Boolean lattice, which by Sperner's Theorem implies that there can be at most $\binom{k-1}{\lceil (k-1)/2 \rceil}$ such schedules. If k is even this matches the claimed bound. If on the other hand k is an odd number, this only gives a bound of $\binom{k-1}{(k-1)/2}$ whereas the claimed bound is $\binom{k-1}{(k+1)/2}$. The last step here, the details of which we leave for the reader to fill in, is to improve the bound using the fact that if we pick some string x with exactly $(k-1)/2$ ones, we cannot also pick the complement of x (by which we mean the string where we keep the first position 0 and flip the remaining $k-1$ bits). Of course, figuring out the details of this is not necessary during the contest: coming up with the construction above, and confidently guessing that it is optimal, is sufficient to solve the problem.

This leads to the following simple algorithm: given n, find the smallest k such that $\binom{k-1}{\lceil k/2 \rceil} \geq n$, enumerate n different binary strings of length k with $\lceil k/2 \rceil$ ones (and starting with zeros), and use these as the schedule.

Problem C: Three Kinds of Dice

Problem authors:
Derek Kisman and Walter Guttmann

Solved by 30 teams.
First solved after 82 minutes.
Shortest team solution: 1303 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

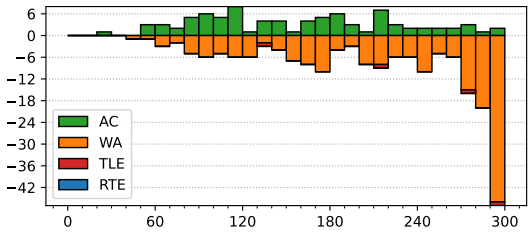
assign weights to those points so that the weighted average of $S_1(v)$ is ≤ 0.5, and the weighted average of $S_2(v)$ is as high as possible (and vice versa).

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 D: Carl's Vacation

Problem authors:
Arnav Sastry and the World Finals judges

Solved by 86 teams.
First solved after 21 minutes.
Shortest team solution: 1537 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:

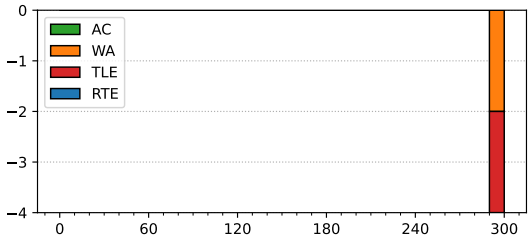
- We iterate over all selections of the faces on both pyramids.
- We rotate the top of the pyramid around the line containing the base of the selected face so that the top of the pyramid lands on the ground.
- We check if the segment between the two rotated tops intersects the two bases, in the right 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 E: A Recurring Problem

Problem authors:
Derek Kisman and Matthias Ruhl

Solved by 0 teams.
Shortest judge solution: 2987 bytes.



This was, in the judges' opinion, the hardest problem in this set. It can be solved using fairly complicated dynamic programming.

You can start by thinking about how to get the first few elements of the generated sequence. The first one is actually quite easy: the number of sequences that start with t is the number of ways of writing t as $\sum_{i=1}^{k} c_i a_i$ for some positive integers k, c_i, a_i.

Figuring out the extensions of this is considerably trickier, because now the c_i's and a_i's interact with each other. In general, suppose you want to count how many ways an order-k sequence can start with $(a_{k+1}, a_{k+2}, \ldots, a_{k+\ell})$. If you know the values of c_k and a_k, then you can essentially subtract the contribution of these and try to continue from there; but you still need to remember the original values of $a_k, \ldots, a_{k+\ell-2}$ because these will be make a contribution multiplied by c_{k-1}.

After some thought, a useful function to compute is the following. For a vector $x = (x_1, \ldots, x_\ell)$ of length ℓ and a vector $a = (a_1, \ldots, a_{\ell-1})$ of length $\ell - 1$, define $f(x, a)$ to be the number of possible choices of r and positive integers $a_0, a_{-1}, \ldots, a_{-(r-1)}$ and $c_1, \ldots, c_r$ such that, for all $1 \leq i \leq \ell$ it holds that

$$x_i = \sum_{j=1}^{r} a_{-r+i+j-1} \cdot c_j$$

Note that, if $a = (x_1, \ldots, x_{\ell-1})$, then $f(x, a)$ counts exactly how many generated sequences start with x. Furthermore, we can define a nice recurrence for f:

$$f(x,a) = \begin{cases} 1 & \text{if } x \text{ is identically } 0 \\ 0 & \text{if } x \text{ has any negative entries} \\ \sum_{a_0,c} f(x - c \cdot (a_0, a_1, \ldots, a_{\ell-1}), (a_0, \ldots, a_{\ell-2})) & \text{otherwise} \end{cases}$$

In the sum, a_0 and c range over "all" positive integers, but can clearly be limited to those such that $x - c \cdot (a_0, \ldots, a_{\ell-1})$ is a non-negative vector.

Ok, so implementing this function f (and adding memoization), we have a way of counting the number of sequences with a given start, but this is still very slow, because as written there are lots of dead-end paths explored, that never lead to us reaching the all-zero vector (and therefore do not contribute anything to the count).

From here, there are at least two approaches for making the solution fast enough:

1. One useful idea for computing f faster is that if $f(x, a) > 0$ then $f(x', a') > 0$ where x' and a' are the vectors of length $\ell - 1$ and $\ell - 2$ where we remove the last element. So when computing $f(x, a)$, we can add a pruning step which first computes $f(x', a')$ and checks that this is positive. This improves the running time dramatically and makes the solution fast enough.

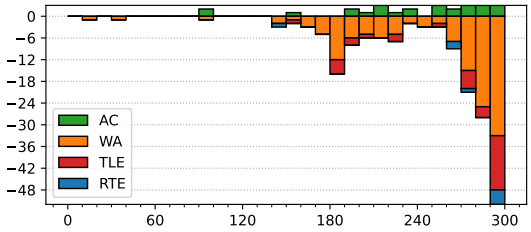
   Furthermore, we can relatively easily change f so that we instead count, given $y = (y_1, \ldots, y_\ell)$, the number of generated sequences with a prefix of the form $(y_1, \ldots, y_{\ell-1}, \leq y_\ell)$. With that in hand we can binary search for the next extension.

   One last idea here is that, once we have honed in on a prefix where there are relatively few (say, less than 500 000 different sequences generating that prefix, we can create all of those sequences and then iteratively evolve them until we find the right one.
2. A cleaner idea is to change the function f to not only count the number of sequences with the given prefix, but also all possible extensions, together with their counts. That way, there is no need to binary search for the next entry and we can generate the prefix one step at a time until we are at the right one.

Problem F: Tilting Tiles

Problem authors:
Paul Wild and Martin Kacer

Solved by 26 teams.
First solved after 90 minutes.
Shortest team solution: 3662 bytes.
Shortest judge solution: 2240 bytes.



Once we have made at least one horizontal and at least one vertical move, the tiles will always be flush in one of the corners of the grid. At this point, we are no longer able to change the outline the tiles to any other shape, other than changing which of the four corners the tiles are anchored in. The only change possible from here is to permute the labels by cycling through these four configurations using the moves right-down-left-up repeatedly. One repetition of these four moves induces a permutation on the tiles, and the configurations reachable from here are determined by the powers of that permutation, plus some extra moves if we want to anchor in another corner.

In total, the reachable configurations are 1) some configurations reachable in 0 or 1 steps that are not anchored in a corner, and 2) some configurations that are all reachable along at most four long cycles that are each determined by a permutation.

If we fix one of these permutations, we are now left with a string theory problem: Given strings s and t, and a permutation p, is there some number n such that after n-fold application of the permutation p to the string s we get the string t?

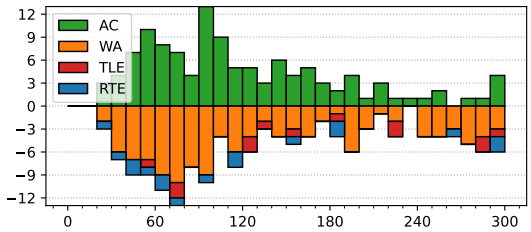
To solve this problem, we decompose the permutation into disjoint cycles, and look at each cycle independently. Using a linear-time string matching algorithm we either find that no solution for that cycle exists, or obtain a requirement in the form of a linear congruence $n \equiv a \pmod{m}$.

Finally, we need to check whether the system of linear congruences has a solution. This is a somewhat standard application of the Chinese Remainder Theorem, but the solution may potentially be huge. Rather than constructing it directly, one can show that it is enough to check each pair of congruences for consistence. As there are only $O(\sqrt{|s|})$ distinct moduli, this too can be done in linear time.

Problem G: Turning Red

Problem authors:
Jakub Onufry Wojtaszczyk and Arnav Sastry

Solved by 117 teams.
First solved after 21 minutes.
Shortest team solution: 1733 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.

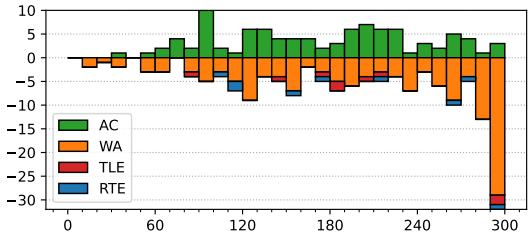
This leads to the following linear-time algorithm: for each connected component of the graph, pick an arbitrary button i, try all 3 possible values $x_i = 0, 1, 2$, and propagate the result. If an inconsistency is found (some equation is not satisfied), then this value of x_i is invalid. Otherwise, check 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 H: Jet Lag

Problem authors:
Jakub Onufry Wojtaszczyk and Federico Glaudo

Solved by 96 teams.
First solved after 33 minutes.
Shortest team solution: 669 bytes.
Shortest judge solution: 246 bytes.



This problem, which has absolutely no real-world inspirations, can be solved with a greedy approach with a short implementation.

One method to solve this problem is to first ignore the restriction that you cannot sleep while rested. This allows you to sleep during some intervals $[e_i, b_{i+1}]$, where $e_0 = 0$. After sleeping during one such interval, we can be awake until $e_i + 2(b_{i+1} - e_i)$, so we greedily sleep during these intervals if they allow us to stay up later than we currently can.

After having a list of sleeping intervals from the above process, we need to modify them such that the beginning of one sleep interval is not in the k minutes after the end of the previous one. To do so, say two consecutive sleep intervals are $[s_i, t_i]$ and $[s_{i+1}, t_{i+1}]$. Then we need

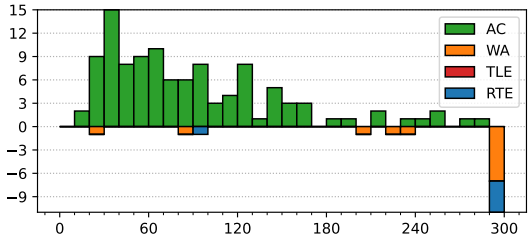
$$s_{i+1} \geq t_i + (t_i - s_i) = 2t_i - s \qquad \Leftrightarrow \qquad \frac{s_{i+1} + s_i}{2} \geq t_i$$

Thus we set t_i to the mean of s_i and s_{i+1} if needed.

Problem I: Waterworld

Problem authors:
Walter Guttmann and Yujie An

Solved by 110 teams.
First solved after 16 minutes.
Shortest team solution: 171 bytes.
Shortest judge solution: 90 bytes.

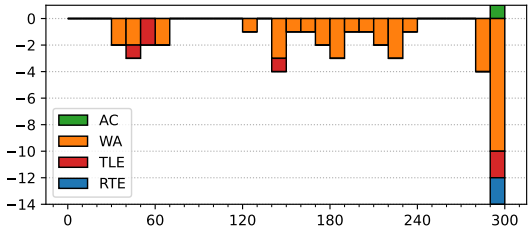


The surface of a spherical cap or a spherical segment is proportional to its height. In cartography, this is known as a cylindrical equal-area projection. Formally, the surface area of any band of height h is $2\pi Rh$ where R is the radius of the sphere. Each horizontal segment covers the same surface area during a whole rotation, and therefore also during a single step. The answer is simply the average of the numbers in the matrix.

Problem J: Bridging the Gap

Problem authors:
Walter Guttmann and Paul Wild

Solved by 1 team.
First solved after 292 minutes.
Shortest team solution: 907 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.

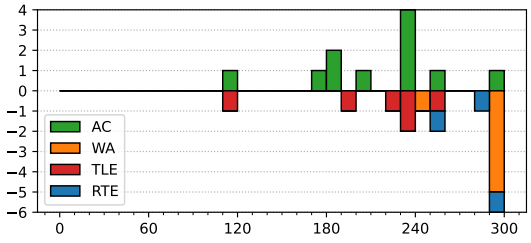
This naturally translates to a dynamic programming solution, where we calculate the cost of having the k slowest people already crossed, and l back-crossings earned. Naively, this is $O(n^3)$ — we have n^2 states, and up to n transitions out of each state. But, if you look closer, it never makes sense 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 K: Alea Iacta Est

Problem authors:
Martin Kacer and Arnav Sastry

Solved by 11 teams.
First solved after 113 minutes.
Shortest team solution: 1459 bytes.
Shortest judge solution: 1511 bytes.



This problem can be viewed as a graph problem and solved with either Dijkstra's algorithm or Bellman-Ford. Alternatively, it may be solved with a value iteration algorithm (which, as far as we could tell, was the approach taken by every single team solving this problem). Here we

describe the first approach.

Consider a system with 7^d states, where each state is a vector $v \in \{1,2,3,4,5,6,*\}^d$, where $v_i \in \{1,2,3,4,5,6\}$ means that the i'th dice is currently showing its v_i'th side, and $v_i = *$ means that we are currently re-rolling the i'th dice. Let us refer to states containing $*$'s as undetermined states, and the remaining states as fixed states.

Initially, we are in the undetermined state $(*,*,\ldots,*)$ corresponding to our first roll, and our goal is to reach one of the fixed states that correspond to a word in the dictionary (after possibly permuting the coordinates).

From a fixed state, we can for a cost of 0 choose to replace any non-empty subset of the coordinates by $*$ and move to that undetermined state, indicating which dice we choose to reroll. From an undetermined state, the $*$'s will be replaced uniformly at random by digits 1-6 and we will move to that fixed state for a cost of 1 (corresponding to the re-rolling of the dice).

Now we want to make the right choices at all the fixed states in such a way that we reach one of the goal states with minimum possible total expected cost. We can do this by running a variant of Dijkstra's algorithm backwards from the goal nodes. Initially, all states x have a distance of $\text{dist}[x] = \infty$, except the goal states which have a distance of 0, and we process the states in increasing order of distance.

Whenever we process a fixed state x, we have to update the distance of all undetermined states y that match x. Suppose that y has a total of $s > 1$ $*$'s, and let D be the set of fixed states matching y that have been processed. Then we can update $\text{dist}[y]$ to $\frac{6^s + \sum_{z \in D} \text{dist}[z]}{|D|}$ (if this is better than the current distance for y); this corresponds to the strategy "keep rerolling these $*$'s until we get to one of the states in D – it takes in expectation $6^s/|D|$ re-rolls to reach a state in D, and since this will be uniformly random its expected distance will be $\frac{1}{|D|}\sum_{x \in D} \text{dist}[z]$. To make this fast, we should keep track of $|D|$ and $\sum_{x \in D} \text{dist}[z]$ for each y and not recompute them every time.

Whenever we process an undetermined state x, the situation is simpler and we can simply update each fixed state y that matches x, updating $\text{dist}[y]$ to $\text{dist}[x]$ (if this is better than current distance for y); this corresponds to the strategy which given y moves to x by changing the corresponding coordinates to $*$.

After running this, the distance for $(*,*,\ldots,*)$ contains the answer.