# Helvetic Coding Contest 2019

Solution Sketches

April 13, 2019

## A. Heidi Learns Hashing

Author: **Damian Straszak (DamianS)**

### A1. Easy

The problem translates to: given a positive integer $r$, find a pair of positive integers $(x, y)$ (with $x$ as small as possible) such that $x^2 + 2xy + x + 1 = r$. One possible approach is to note that necessarily $x \leq \sqrt{r} \leq 10^6$ and thus it is fine to test every possible $x$ and find the corresponding $y$ (from a univariate linear equation).

Alternatively, one can observe that if $r$ is even then there is no solution and if $r$ is odd then $(x, y) = (1, \frac{r-3}{2})$ always works (as long as $y$ is positive)!

### A2. Medium

Let us first find a method to check whether a shift by a fixed number $k$ yields a solution or not. For the sake of simplicity let us work with the case of $n = 12$, let also $k = 3$. We can start by writing a set of equations (all modulo 2)

$$(1) \qquad \begin{cases} x_0 + x_3 &= y_0 \\ x_3 + x_6 &= y_3 \\ x_6 + x_9 &= y_6 \\ x_9 + x_0 &= y_9 \end{cases}$$

Note that for $x$ to be solution to the shift equation, these equations are necessary to be satisfied, but of course it is not yet sufficient. Also it is easy to see that the 3 first equations can be always satisfied, and that by taking the sum of all these equations we obtain

$$0 = y_0 + y_3 + y_6 + y_9$$

which means that $y_0 + y_3 + y_6 + y_9$ should be even. This can be easily seen to be a necessary and sufficient condition for (1) to have a solution $x$. Similarly we can write an analogous system for bits $x_1, x_4, x_7, x_{10}$

$$(2) \qquad \begin{cases} x_1 + x_4 &= y_1 \\ x_4 + x_7 &= y_4 \\ x_7 + x_{10} &= y_7 \\ x_{10} + x_1 &= y_{10} \end{cases}$$

which is solvable iff $y_1 + y_4 + y_7 + y_{10}$ is even. Finally we obtain that there is a solution $x$ to the $k$-xor-shift equation iff

$$\begin{cases} y_0 + y_3 + y_6 + y_9 & = 0 \\ y_1 + y_4 + y_7 + y_{10} & = 0 \\ y_2 + y_5 + y_8 + y_{11} & = 0 \end{cases} \tag{3}$$

Note also that the set of equations corresponding to the value $k = 9$ would be exactly the same as above! This observation is quite easy to generalize for all values of $n$ and $k$. More precisely, whenever $\gcd(n, k_1) = \gcd(n, k_2) = d$ then one can write $d$ equations for $y$ that determine whether the xor-shift equations has a solution for $k_1$ (and equivalently for $k_2$).

The complete solution is: first precompute the answer for every $k$ being a divisor of $n$, then for every other $k$, compute $\gcd(n, k)$ to reduce to one of the precomputed cases. The complexity is $O(n \cdot d(n)) = O(n^{3/2})$ where $d(n)$ is the number of divisors of $n$.

## A3. Hard

This problem can be cast simply as: given a polynomial $f(x) = \sum_{i=0}^{n-1} f_i x^i$ with integer coefficients, find a large prime $p$ and an $r \in [2, p-2]$ such that $f(r) = 0 \pmod{p}$.

One possible approach would be to pick a prime $p$ at random and check all values of $r$ one by one. Since for a random value of $r$ the value $f(r) \pmod{p}$ can be considered random, we expect that the probability that any $r$ is good is roughly

$$\left(1 - \frac{1}{p}\right)^p \approx e^{-1} \geq \frac{1}{3}.$$

Therefore we need to repeat this only a constant number of times to be successful. However, the issue is that this requires $\approx p$ evaluations of the polynomial and thus around $O(np)$ time, which can be made $O(p^2 + n)$ if $n$ is large. Still, it is too much as we are talking about $p \approx 10^5$.

One possible approach here is to use an algorithm based on FFT to make all these evaluations in time $O(p \log^2 p)$. This is indeed enough to solve the problem, however it is certainly not the simplest, nor the most elegant :), way to do that.

Another idea would be as follows. Let $d$ be any divisor of $p - 1$, it is a basic fact about the finite field $\mathbb{F}_p$ that there exist exactly $d$ elements $r$ in $\mathbb{F}_p$ such that

$$r^d = 1 \pmod{p}.$$

Note that if $r$ is such an element, and say $d = 3$ then for instance

$$r^5 + r^{10} = r^{3+2} + r^{3 \cdot 3+1} = r^2 \cdot r^3 + r \cdot (r^3)^3 = r^2 + r \pmod{p}.$$

In fact by applying this trick to the whole polynomial we obtain a reduced polynomial

$$f^{(d)}(r) = \sum_{i=0}^{d-1} \alpha_i r^i.$$

Which can be evaluated in $d$ points in time $O(d^2)$.

The algorithm becomes now to pick a small $d$, pick a prime of the form $p = ld + 1$ and then find all $r$'s such that $r^d = 1 \pmod{p}$ and evaluate the polynomial $f^{(d)}$ on all these inputs, hoping that one of them gives 0. The crucial trick is that since $f^{(d)}$ can be computed once and for all, we can test a number of primes and have $d$ candidates for each to test in $O(d^2)$ time. Note that if we still believe in the randomness of the output of $f^{(d)} \pmod{p}$ (and we certainly should!) then roughly $\frac{m}{d} \approx \frac{10^5}{d}$ primes should be enough.

There are two remaining issues: (1) which $d$ to pick and (2) how to find $r$'s that satisfy $r^d = 1 \pmod{p}$. It turns out that the best choice for $d$ are small odd primes $d = 3, 5, 7, 11, 13$, etc. (in fact it is good to test

all of them) so that we still have a lot of primes of the form $p = ld + 1$ to test and for $O(d^2)$ to not be too expensive. With such a choice of $d$ it is also not too hard to find the solutions to $r^d = 1 \pmod p$. More specifically the following algorithm works quite well:

1. Given $d$ (prime) and a prime $p = ld + 1$,
2. Sample a random element $r \in \{1, 2, \dots, p-1\}$,
3. If $r^{\frac{p-1}{d}} \neq 1$ then go to the next step, else go back to 2.
4. Output $r, r^2, r^3, \dots, r^d \pmod p$.

It is a not too hard exercise in algebra to see that the above algorithm indeed outputs all desired solutions and furthermore works in expected time $O(\log p)$.

## B. The Doctor Meets Vader

Author: **Wajeb Saab (Wajeb)**

### B1. Easy

We are given $s$ spaceships, each with an attacking power $a_i$, and $b$ bases, each with a defensive power $d_j$ and gold $g_j$.

We are asked to output for each spaceship, the sum of the gold of the bases it can attack, that is: $res(i) = \sum g_j, \forall j : a_i \geq d_j$

An $O(sb)$ solution would not pass the time limit. Instead the intended solution is to sort the bases by their defensive power, and the ships by their attacking power, in increasing order. Then, we can use the fact that if a weaker ship can attack a base, so can a stronger ship. This way we only have to iterate once over all the bases.

The overall complexity reduces to $O(s \log s + b \log b)$.

Another solution of that would pass the time limit would be to sort the bases in increasing order of their defensive strength. Then, compute the prefix sum of the gold (that is, prefix[$j$] would hold the sum of gold of all the bases from 0...$j$). Finally, for each ship, perform a binary search over the prefix sum and the bases arrays to compute how much gold it could steal.

Overall complexity: $O(s \log b + b \log b)$.

### B2. Medium

We are given:

- an undirected graph with $n$ nodes and $e$ edges
- $s$ spaceships: attack $a_i$, location on graph $x_i$, and fuel $f_i$
- $b$ bases: defense $d_j$, and location on graph $x_j$

Using the constraints, we can build a bipartite graph with spaceships on the left and bases on the right. The edges would connect ships to the bases that they can attack. This can be checked by pre-computing the all-pair shortest path between any two nodes (in $O(n^3)$), then performing a brute force check between all ships and bases (in $O(sb)$).

When a base is attacked, we lose $k$ gold. But, we can create dummy bases that would lose us $h$ gold instead. These bases are connected to all ships in the bipartite graph. Moreover, we are given the fact that dummy bases will always be attacked. That is, they will always be part of a maximum matching.

The problem reduces to performing maximum matching on the bipartite graph, and checking how many dummy bases to build in order to minimize the amount of gold lost. However, trying out all cases is not guaranteed to pass the time limit.

Instead, the idea is to realize that the optimal solution always requires either building $s$ dummy bases or zero dummy bases.

Consider the case when $h > k$, in such a case it does not make sense to build any dummy base, because it would cost more to build it than it would save.

When $h < k$, the first few dummy bases we build might cost us extra gold, because they will be matched with ships that weren't attacking any bases earlier. After we build enough dummy bases, they will start decreasing our cost, by taking away ships from actual bases (which are more expensive).

Let $g(x)$ be the gold lost when $x$ dummy bases are built. The function of gold lost starts at $g(0)$, increases until some point, then decreases to reach $g(s)$. Building dummy bases beyond $g(s)$ doesn't make sense.

Therefore, the optimal solution is the minimum of $g(0)$ and $g(s)$. $g(s)$ is simply $sh$, whereas $g(0)$ is $mb$, where $m$ is the maximum matching obtained from the original bipartite graph.

Any matching algorithm used would pass the time limit.

Total complexity: $O(n^3 + sb\sqrt{s+b})$.

## B3. Hard

We are given:

- an undirected graph with $n$ nodes and $m$ edges
- $s$ spaceships: attack $a_i$, location on graph $x_i$, fuel $f_i$, and operating price $p_i$
- $b$ bases: defense $d_j$, location on graph $x_j$, and gold $g_i$

We are also given $k$ dependencies between ships, and are asked to compute the maximum profit when operating the optimal subset of ships.

We first note that the problem can be divided into two independent parts. In the first part, we compute the maximum profit achievable by each ship. In the second part, we decide which subset of ships to use based on the dependencies.

For the first part, we can first compute the all-pair shortest path in $O(n^3)$. Then, for each node in the graph, we build a data structure that allows us to access the best base to attack in that node given a certain ship's attacking strength. Building and accessing this data structure can be done in $O(sn \log b + b \log b)$.

We must be careful to assign a negative infinity profit to ships that cannot be operated at all, as we will see in the next part of this problem.

Now, we know for each ship, the best profit it can achieve. From there, we can always take the ships with positive profits that don't appear in the dependencies, and never consider the ships with negative profits that don't appear in the dependencies.

This leaves us with $\ell \leq 2k$ ships that appear in the dependency list. For these ships, we can build a dependency graph as follows:

1. Create a dummy node $S$
2. For each ship $s_i$ with positive profit: Add a directed edge from $S$ to $s_i$, with a weight equal to that profit
3. Create a dummy node $T$
4. For each ship $s_i$ with negative profit: Add a directed edge from $s_i$ to $T$, with a weight equal to the absolute value of that profit
5. For each dependency $s_i$ to $S_j$: Add a directed edge from $s_i$ to $s_j$ with infinite weight

It can be shown that the maximum profit that can be achieved is equal to the sum of all positive profits minus the weight of the minimum $S$-$T$ cut in the dependency graph.

As the number of nodes in the dependency graph is $O(k)$, any maximum flow algorithm would pass the time limit.

Brief sketch of why the minimum cut achieves the solution: The idea is that all the ships on the $S$-side of the cut would be taken. It can be seen that it is impossible to take a ship without its dependency, as that would involve performing a cut over an infinite edge. It can also be seen that the ships that cannot be operated will never be on the $S$-side of the cut, because they are connected to $T$ with an infinite edge. From what remains, since the positive profit ships are connected to $S$, and the negative profit ships are connected to $T$, then the cut includes all the POSITIVE ships that are NOT INCLUDED, and all the NEGATIVE ships that are INCLUDED. Therefore, summing all positive ships and subtracting the minimum cut yields the intended solution.

## C. Heidi and the Turing Test

Author: **Chia-An Yu (esrever)**

### C1. Easy

The input size of this problem is set so that it is feasible to check all possible squares (less than $50 \times 50 \times 50$ possibilities). For each square, it takes $O(4n)$ time to check if it satisfies the constraint and identify the point that is not on the boundary. This gives a solution with time complexity $O(50^3 \times 4n)$.

There is a faster solution. Notice that the coordinates of the sides of the square will appear at least two times because $n \geq 2$. Therefore, since there is only one point not on the boundary, taking the maximum among $x$-coordinates which appear at least twice will give us the $x$-coordinate of the right side of the square. The other three sides can be obtained similarly with different combinations of maximum/minimum and $x$-/$y$-coordinates. After knowing the sides of the square, it is easy to identify the point not on the boundary. This solution has time complexity $O(n)$.

### C2. Medium

The first step is to rotate the plane by 45° using the mapping $(x, y) \mapsto (x + y, x - y)$. Then the problem becomes finding an axis-aligned square with side length $2r$ that covers as many points as possible. The second step is to replace each point $(x, y)$ with a square having lower-left corner at $(x, y)$ and upper-right corner at $(x + 2r, y + 2r)$. This transforms the problem into a classic exercise for sweeping line and segment tree: given many axis-aligned rectangles (squares in our case), find a point that is covered by the most rectangles.

The solution proceeds as follow. Initially, we have an empty real line. We use a sweeping line going from bottom to top. Whenever the line meets a horizontal side of a rectangle with lower-left corner $(x_l, y_l)$ and upper-right corner $(x_r, y_r)$, cover the segment $[x_l, x_r]$ if it is a lower side and remove it if it is an upper side. When we cover a segment, we find out the maximum overlapping of all segments on the line. The answer is then the maximum overlapping that ever happens as the sweeping line goes up. In order to efficiently cover / remove segments and query for the maximum overlapping, we can use a segment tree to do each operation in logarithm time.

The overall time complexity of the solution is $O(n \log(R))$, where $R$ is the range of the coordinates. While we can improve it to $O(n \log(n))$ by discretizing the coordinates, using the original coordinates is enough to pass the tests.

## C3. Hard

As you may guess from the name of the variable in the input, the intended solutions are variations of $k$-means clustering algorithm.

Let's start with the simple case of one ring, *i.e.* $k = 1$. Our solution first solves a linear least-squares problem

$$\begin{bmatrix} 2x_1 & 2y_1 & 1 \\ \vdots & \vdots & \vdots \\ 2x_n & 2y_n & 1 \end{bmatrix} \begin{bmatrix} o_x \\ o_y \\ z \end{bmatrix} = \begin{bmatrix} x_1^2 + y_1^2 \\ \vdots \\ x_n^2 + y_n^2 \end{bmatrix}, \tag{4}$$

where $(x_i, y_i)$ are the sampled points. Then the ring is estimated by the circumcenter $(o_x, o_y)$ and the radius $\sqrt{z + o_x^2 + o_y^2}$. This method works well even for $n$ as small as five (given the noise in this problem). Note that when there is no noise, this gives exactly the circumscribed circle of these points and only three points is needed to solve the system. The time complexity for solving this least-squares problem is $O(n)$.

There are many other ways to solve this one ring case. For example, we can define an objective function

$$f(o_x, o_y, r) = \frac{1}{n} \sum_i \left| (o_x - x_i)^2 + (o_y - y_i)^2 - r^2 \right| \tag{5}$$

and solve for its minimum using algorithm such as gradient descent. One can also try to find $(o_x, o_y)$ that minimizes the ratio (the distance to the furthest point) / (the distance to the closest point).

For multiple rings, we use an iterative algorithm similar to $k$-means algorithm. Initially, each point is assigned randomly to one of the $k$ groups. Then we repeat the following two steps until a stopping criteria is met:

- For each group, randomly pick five points and solve (4) to obtain a ring.
- For each point, reassign its group to the closest ring if such distance is less than $0.2\times$(radius of that ring). Otherwise, we assign it to no group. The distance between a point and a ring is the minimum distance between the point and any point on that ring.

The algorithm stops when every point belongs to a group after the reassignment.

The ideas of the two steps are similar to the ones in $k$-means algorithm. However, it doesn't work if we directly translate them into estimating rings (using all points) and reassigning naively. The modification is important and ensures that the estimated rings are close to the real ones with high enough probability, which means that the assignment converges to the correct one fast enough and that the algorithm stops early enough.

The time complexity for the first step is $O(k)$ with the time to solve the small linear system (4) as a constant factor. For the second step, we need to compute the distance between every point and every group, so the time complexity is $O(nk^2)$. Our solution stops within 2000 iterations with a random reassignment after every 200 iterations.

Another solution we have also uses $k$-means algorithm. The first step is to compute many candidate rings. A candidate ring is the circumscribed circle of three random sampled points. A candidate ring should also satisfy the constraints we know from the problem: the range of the radius and the center, the number of points that are close to the ring, etc. It is very unlikely that some of the candidate rings match the real ones immediately due to the noise. However, their center will form clusters around the true centers. Therefore, we can use the ordinary $k$-means algorithm to estimate the centers, and the radii are estimated by taking the average of the radius of the candidates in the same cluster.

Here's the estimation of the running time of this solution. The time complexity of getting one candidate is

$$\frac{1}{\text{the probability of getting one candidate}} \times O(nk)\,, \tag{6}$$

where $O(nk)$ is the linear scan to check for the constraints. The probability of getting one candidate ring can be estimated as follow. Firstly, the three points should be from the same ring, which has the probability of roughly $\frac{1}{k^2}$. Also, the points need to be well separated so that the affect of the noise is limited on the distance between the circumcenter and the true center. Let's define the "well separation" this way: we divide the ring uniformly into six sectors, and the points are well separated if they fall into three sectors that are not adjacent to each others. The probability of this happening is $\frac{12}{6^3} = \frac{1}{18}$, given how the points are sampled from the ring. This solution uses less than 2000 candidates, so the time complexity for computing all candidates is $O(2000 \times 18nk^3)$. To estimate the rings, it runs 100 times $k$-means algorithm with 100 iterations each, where each iteration takes $O(\text{number of candidate rings})$.

# D. Parallel Universes

Author: **Zhejiang University (ZOJ)**

## D1. Easy

This is a simple simulation problem. Assuming that the length is $l$ and Doctor's position is $k$, we can have the following cases:

- A universe is created at some position after $k$. In this case we increase $l$ by one.
- A universe is created at some position before $k$. In this case we increase both $l$ and $k$ by one.
- A link is broken at some position after $t_1 \geq k$. Set $l$ to $t_1$ and keep $k$ the same.
- A link is broken at some position $t_2 < k$. In this case we reduce both $l$ and $k$ by $t_2$

Clearly the operations above can be performed in linear time so the total complexity is $O(t)$.

## D2. Hard

First notice that $k = 1$ and $k = n$ are special cases and for them we should output $n$ as a result. Now suppose we are given some $2 \leq k \leq n$ and let $x_{i,j}$ represent expected length if we start at chain of length $i$ at position $j$.

Let us first present the slow $\Theta(m^6)$ solution. In the most general form we can create a system of equations for $x_{i,j}$ ($3 \leq i \leq m$, $2 \leq j \leq m-1$) as follows:

$$x_{i,j} = \sum_{i',j'} p((i,j),(i',j'))\, x_{i',j'}$$

where $(i', j')$ go over all states reachable from the state $(i, j)$ and $p((i,j),(i',j'))$ represents the transitions probabilities. Furthermore, we know that $x_{i,1} = x_{i,i} = i$ for each $1 \leq i \leq m$. Now let us look at the all possible transitions and the corresponding contributions:

- Inserting a universe at a position after $j$. This happens with probability $(1 - \frac{i}{m})\, \frac{i+1-j}{i+1}$ and the next state is $(i+1, j)$. Contribution is thus $(1 - \frac{i}{m})\, \frac{i+1-j}{i+1}\, x_{i+1,j}$.
- Inserting a universe at some position before $j$ with probability $(1 - \frac{i}{m})\, \frac{j}{i+1}$. This way we end up at the state $(i + 1, j + 1)$ and the resulting contribution is $(1 - \frac{i}{m})\, \frac{j}{i+1}\, x_{j+1,i+1}$.
- Breaking a link at some positions between $j$ and $i$. The new chain has smaller length ($i$ decreases) but the position we end up at remains the same. Formally, the total contribution is $\frac{i}{m} \sum_{i'=j}^{i-1} \frac{1}{i-1}\, x_{i',j}$.

- Breaking a link at position between 1 and $j$. The new chain has smaller length and the smaller position. Total contribution in this case is $\frac{i}{m}\sum_{t=1}^{j-1}\frac{1}{i-1}\,x_{i-t,j-t}$.

This way we have obtained a system of $O(m^2)$ equations with $O(m^2)$ unknowns and the Gaussian elimination leads to a $O(m^6)$ solution.

Now, the trick is to consider the equations in a cleverly chosen order so that we end up with a $O(m)$ equations and variables. Looking at the above equations we see that $x_{3,2}$ can be represented as a function of $x_{4,2}$, $x_{4,3}$ and some known constants. We can represent $x_{4,2}$ as a function of $x_{5,2}$, $x_{5,3}$ and $x_{2,2}$ which is already a function of $x_{4,2}$ and $x_{4,3}$. Continuing in this fashion, we end up with variables $\{x_{m,j}\}_{j=2}^{m-1}$ and $O(m)$ equations. If we represent each $x_{i,j}$ as a linear combination of $\{x_{m,j}\}_{j=2}^{m-1}$, we will need $O(m)$ fields for each of them. Notice that, since $x_{i,j}$ depends on $O(m)$ previous variables and each of them has $O(m)$ fields, we need to keep the prefix sums in order to speed up the calculations. Gaussian elimination on this system takes $O(m^3)$ operations. After we know the variables $x_{m,j}$, we can back-propagate them in order to compute $x_{n,k}$.

# E. Daleks' Invasion

Author: **Jakub Tarnawski (dj3500)**

## E1. Easy

We are given an edge-weighted undirected graph and, for some edge $c$, want to find the maximum weight $e$ such that if the weight of the edge is changed to $e$, this edge would still be in some minimum spanning tree (MST).

One way would be to simply try all possible weights – however, that would be too slow, even we make the observation that we only need to try the numbers that are weights of other edges. However, we can perform a binary search on the result $e$; to check, for a given weight $e$, whether $c$ finds its way into an MST, we just build an MST using a fast algorithm such as ones by Dijkstra-Prim or Kruskal. We should take care to include $c$ in the solution if it is tied for weight with other edges, as there should exist *some* MST that $c$ is part of. If we use Kruskal's algorithm and sort edges once (before the binary search), we will get a runtime of $O(m \log n \log^* n)$.

Alternatively, we can run Kruskal's algorithm just once, but disregarding $c$. The weight we want is the one from the time when the algorithm joins the components containing the endpoints of $c$. Thus, after every merge, we check whether the endpoints of $c$ are now in the same component, and if yes, we output the weight of the edge we just added. If the algorithm never joins the endpoints of $c$, this means that $c$ is a bridge and we should print $10^9$.

## E2. Medium

Now we need to solve the task for all non-tree edges (the MST is guaranteed to be unique). We claim that for a non-tree edge $(u, v)$, the correct answer is the largest weight on the path between $u$ and $v$ in the MST. (Proof: $\leq$: this follows from the cycle property of MSTs; $\geq$: if the weight of $(u, v)$ is larger than this maximum, then swapping the edge attaining the maximum with $(u, v)$ would yield a cheaper tree, a contradiction.) There are many ways to compute all such values; here we present one that is a generalization of the binary-search solution to the Easy version.

Namely, we can do binary search on all non-tree edges in parallel! For each non-tree edge, maintain the interval of values still to be checked. Run Kruskal's algorithm $O(\log n)$ times (or, more simply, until all of the above intervals are empty). Before each run, build an array of vectors $v[m]$, where $v[m]$ holds those edges for which the middle of the interval falls at $m$. Then, while running Kruskal, after having processed the $i$-th edge, for each $c \in v[i]$ check whether the endpoints of $c$ are united yet, and shrink the interval accordingly.

### E3. Hard

Now we need to solve the task also for tree edges. First we find an MST $T$ and determine the answer for the non-tree edges as in Medium. The MST is no longer guaranteed to be unique, but it doesn't matter.

For starters, let us figure out what we are trying to compute. For every tree edge $(u, v)$ we consider the cut $(S, V \setminus S)$ that is induced by the cut $\{(u, v)\}$ in the tree. To put this less obliquely, let $S$ be the vertices reachable from $u$ in the graph $T - \{(u, v)\}$.

**Claim 1** *The answer for* $(u, v)$ *is the minimum weight of an edge in this cut that is not* $(u, v)$*, i.e.,* $\min_{c \in E(S, V \setminus S) \setminus \{(u,v)\}} \textit{weight}(c)$.

**Proof:** $\geq$: as long as the weight of $(u, v)$ is minimum in this cut, $(u, v)$ is in some MST (this is called the cut property of MSTs).
$\leq$: suppose that the weight of $(u, v)$ is larger than this minimum; we will show that $(u, v)$ is not in any MST. Let this minimum be called $M$ and be attained for an edge $c'$. The edge $c'$ is not in $T$ (because in $T$ the only edge in this cut is $(u, v)$); let us consider the path in $T$ between the endpoints of $e'$. This path uses, apart from $(u, v)$, only edges of weight at most $M$ (for otherwise we could swap $c'$ for some edge of weight larger than $M$, which would prove that $T$ is not optimum after all). This path plus $c'$ is a cycle on which the only edge of weight larger than $M$ is $(u, v)$. Now, the cycle property of MSTs implies that $(u, v)$ is not in any MST. ■

So every non-tree edge $(a, b)$ constitutes an upper bound on the answer for all tree edges on the path between $a$ and $b$ in $T$. For every tree edge we want to compute the minimum of these upper bounds. How do we do this? Let us break up every path from $a$ to $b$ into two paths: $a$ to $\text{lca}(a, b)$ and $b$ to $\text{lca}(a, b)$, where $\text{lca}(a, b)$ is the Lowest Common Ancestor of $a$ and $b$ in $T$. Every such path can be covered with $O(\log n)$ paths of lengths that are powers of two (just like in the design of the $O(n \log n)$ LCA algorithm). In every vertex, we store the best-seen-so-far bounds on the paths from this vertex to its $2^k$-th ancestor, for every $k$. Once all non-tree edges are processed, we can extract the best bounds for every tree edge from this structure. As an aside, the Medium problem can also be solved using a similar method.

---

## X. The Cybermen Moonbase

Author: **Buddhima Gamlath (bgamlath)**

This problem was not featured in the online mirror, but can be solved on CodeForces Gym (`https://codeforces.com/gym/102271`).

### X1. Easy

In this problem, you are required find the number of paths from the left edge to the right edge on a grid with obstacles moving up and down. If the obstacles are *fixed*, there is a simple dynamic program solution: If `dp[x][y]` denote the number of ways to reach cell $(x, y)$ modulo $M$ (where $M = 1000000007$) we have

$$\texttt{dp[1][y]} = \begin{cases} 1 & \text{If initially there is no obstacle at } (1, i) \\ 0 & \text{Otherwise,} \end{cases}$$

and, for $1 < c \leq W$, we have

$$\texttt{dp[c][y]} = \begin{cases} \left( \sum_{\substack{1 \leq y' \leq h \\ |y' - y| \leq k}} \texttt{dp[c-1][y']} \right) \mod M & \text{If there is no obstacle at } (1, i) \\ 0 & \text{Otherwise.} \end{cases}$$

The running time of this dynamic programming implementations is $O(H \cdot W \cdot K)$.

But how can we do this all those obstacles up and down? It is easy if you notice that, since the TARDIS advances by one column at a time step, we only need to compute the position of obstacles in column $c$ only for the time step $c$. If you count the difference between number of $U$'s and $D$s in the first $c-1$ characters of $S$, then you know by exactly how much you need to shift the obstacles in $c$ the column to compute their position by the time TARDIS reaches $c$'th column.

Once you compute the array `dp`, the solution is just the sum of entries in its last column modulo $M$.

## X2. Hard

Here, the problem is to find a path in a grid that does not collide with any cannonball moving in a horizontal or vertical direction. The difficulty here is that the cannonballs collide with each other and destroy themselves. Thus it is possible that some paths that are not valid if there were no collisions among cannonballs could become valid when we take collisions into consideration. Therefore, what we need is to figure out which cannonballs collide at which time, and we need to do this in order. This is because if two cannonballs collide at time $t$ first, then they can no longer collide with any other cannonball after time $t$.

An easy way to do this is consider each pair of cannonballs and determine when they collide. Then we can sort them by the collision time and process the collisions in order, discarding pairs where at least one of the cannonballs have already collided previously with another. However this approach is too slow for the given time constraints as it would need $O(N^2)$ time.

Now, how can we find the collisions efficiently? For this, let's consider all cannon balls in an extended grid, where we assume that all cannons fire at time 0, but they are not at the edge of the boundary, but are appropriately faraway.

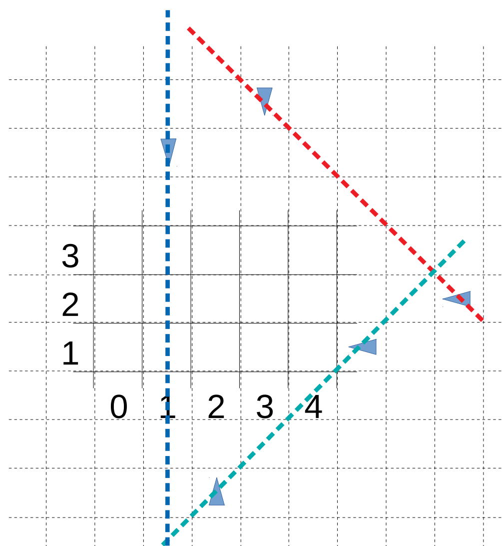


Figure 1: An example of an extended cannon field.

To give an example, consider a $3 \times 4$ grid with the following cannon fires: `1 L 1`, `3 L 2`, `2 D 1`, `3 D 3`, `3 U 2`. Then, as shown in Figure 1, for the first cannon fire, we can assume a cannonball moving left is on (5,1) at time 0 so that it will be at (4,1) at time 1. Similarly, we can assume that a left-moving cannonball is on (7,2) at time 0 so that it will be at (4,2) at time 3, and so on for other cannonballs.

It is easy to see that, if a left-moving cannon ball collides with a one that is moving down, then both of them are in a line that has a slope of negative one (e.g., the red line in Figure 1). Similarly, for a collision between a left-moving cannonball and a one going up, both of the cannonballs must be in a line with slope

one (e.g., the green line), and for an up and down collision, they must be in the same vertical line. Thus we now only have to consider the collisions in those lines.

We can then use a priority queue to consider the possible collisions in order and find exactly which cannonballs collide at what times. Once we know this information, we can find which cells are inaccessible for the TARDIS. For example, if TARDIS collides with a cannonball moving up or down at column $c$, the it has to at time $c$ (because TARDIS advances one column at each time step). Thus for up or down moving cannonballs in column $c$, we can find their exact position at time $c$. If such a cannonball has not collided with any other cannonball before time $c$, then TARDIS cannot be at the same cell as this cannon ball at time $c$ on column $c$. For left moving cannons, we have to consider both types of collisions, but this not so difficult. The tricky part is that, with left moving cannonballs, some cells are inaccessible to TARDIS from any direction, while some other cells are inaccessible from left, but accessible from diagonal directions (I.e., if TARDIS tries to access these cells from left, it would result in a head-on collision of type 2).

Once we have figured out which moves are invalid for TARDIS, we can find a path from $(0, S)$ to $(W, E)$ with a simple dynamic program (if there exist any).