Helvetic Coding Contest 2018

Solution Sketches

March 17, 2018

A. Death Stars

Author: **Andrii Maksai**

A1. Easy

Here you were asked if two square matrices can be aligned by rotation and vertical / horizontal flips. Since rotating the matrix 4 times brings it back to the original state, and so do two vertical or horizontal flips, one had to notice that there aren't many possible matrices that could be obtained by such actions from the initial one. Therefore we can go through all matrices we could possibly obtain from the first one, and check if any of them is equal to the second one. The constraints allowed to do this in a naive way in $O(n^2)$ time. There are only 8 possible matrices that we can obtain from a matrix by rotation and flipping – one might assume there are 16 (rotation by 0, 90, 180, 270 degrees $\times$ 0 or 1 vertical flip $\times$ 0 or 1 horizontal flip), but some of them are actually the same. Rotations and flips could be also done in a naive way in $O(n^2)$.

A2. Medium

In this problem you were asked to find a common $m \times m$ submatrix of two matrices of sizes $n \times m$ and $m \times n$. The naive solution is to go through all possible $m \times m$ submatrices of the first, and second matrix, and check if they are the same. There are $n - m + 1$ possible submatrices of size $m \times m$ of a matrix of size $n \times m$ - those spanning 1-st to m-th row, from 2-nd to $m + 1$-st row, $\cdots$, from $n - m + 1$-st to n-th row. We can also naively check if two submatrices are equal in $O(n^2)$ by comparing all elements. This yields a $O(n^2m^2)$ solution, since $O(n - m + 1)$ is same as $O(n)$ for this problem, since $m \leq n$. However, this was (hopefully) too slow to pass the time limit. To speed things up, one needed a way to compare two $m \times m$ submatrices faster. To do so, we will compare the rows of the submatrices in $O(1)$, thus bringing the total comparison time to $O(m)$.

One way to do that is to compute the hash function of every $1 \times m$ submatrix in the first and the second matrix, and compare hashes instead of the rows. One possible hash-function for such a submatrix, which we would treat as string s, would be a polynomial hash $H(s) = (\sum_{i=1}^{|s|} s[i]b^i) \bmod P$, where b is some number larger than any $s[i]$, and P is a prime number. If the number was not prime, chances for hash-collisions increased significantly. Comparing hashes for m rows takes $O(m)$, and total running time would therefore be $O(n^2m)$.

Another way to do that is to use the Knuth-Morris-Pratt or any other algorithm for fast string matching. It allows to find all occurences of a row of the first matrix in the second matrix in $O(n + m)$, so we can precompute all occurences for all n rows of the first matrix and all m rows of the second in $O(nm(n+m)) = O(n^2m)$. After that, we can once again compare submatrices by comparing whether the rows are equal, using our precomputed values in $O(m)$, thus keeping the total running time $O(n^2m)$.

A3. Hard

This problem asked you to find out N common points in two sets that were rotated and translated with respect to each other, and additional noise points were added to each set. The problem can be split in two parts: finding the rotation and translation to align the points, and finding correspondences in the aligned sets of points.

First part, finding rotation and translation to align the points, has several possible solutions, with the key observation being that we have a very large number of points.

One solution observes that since there are not so many noise points (at most half of the real points), but many points in general, there will be at least two real points that both belong to the convex hull of the first and the second set of points. One can check that the expected number of points on the convex hull, even for the smallest value of N, is large enough so that even with the highest proportion of noise (33%) the probability that this doesn't happen is quite low. Once the corresponding two pairs have been found, we can find the rotation and translation so as to align the endpoints of the corresponding segments. Note that there will be two possible options to align them ((A, B) to (C, D) and (A, B) to (D, C)), so we will need to check two options in the second part of the solution. The running time of this part is $O(N \log N)$ for computing the convex hull, and negligible for comparing all pairs on the convex hulls.

Another solution observes that our points, generated originally in the square with sides 20000×20000, will still lie in some rotated and translated square after rotation and translation. They will also fill the square almost completely, allowing us to find the rotation and translation that brings it back to the square with center at origin and sides parallel to the axes. Once we do that for both sets of points, points there are going to be aligned. To find the rotation, one could, for example, perform a ternary search on the angle α such that, once the points are projected onto line drawn at the angle of α, distance between the minimum and the maximum point of the projection is minimized. One should note though that there will be four possible options for the best rotation angle, since we can rotate square by 0, 90, 180, 270 degrees and have the sides parallel to the axes, so we will need to check all those options in the second part of the solution. The running time of this part is $O(N \log P)$, where P is the precision of our ternary search.

In the second part, we want to find for each point from the first set the closest one from the second set, and pick N pairs with the shortest distances. Note that we can not assume the points to be aligned perfectly for multiple reasons – firstly, they are given in the input with 2 digits of precision; secondly, finding the rotation using ternary search introduces some degree of imprecision; and thirdly, computations in floating-point numbers are imprecise. Finally, if we consider several options for rotation and translation, we need to pick the option in which the sum of N shortest distances is the shortest – this will be the correct rotation and translation, and in all other cases we expect the sum of distances to be much larger.

To find the closest point in the second set for each point in the first set, several heuristics can be used. First one is to project all points onto a random line, and sort them by the projection coordinate. Since we expect aligned points to be close, their projections will also be close. Therefore, we can afterwards check, for each point of the first set, several points that are closest on the projection, and pick the one that is closest on the plane. Since points were generated randomly, it is very unlikely that there are going to be many points that are closer in projection that the actual closest point – in particular, we used $K = 150$ closest points on the projection. The running time of this solution is $O(N \log N + NK)$, where the first term comes from sorting points by the projection coordinate, and the second term comes from looking at every K neighbours of every point in the first set.

Another possibility is to discretize the plane by splitting it into K horizontal and vertical stripes, a total of K^2 squares, and for each point, check all the points in the corresponding square and its neighbours. As long as squares are large enough that the closest point from the second set cannot be anywhere by in the neighbouring squares, this solution should work. This solution takes K^2 memory, and has a running time of $O(N \cdot \frac{N}{K^2})$ memory, since each of the K^2 squares on average will contain $\frac{N}{K^2}$ points, so picking $K = \sqrt{N}$ yields running time and memory consumption of $O(N)$.

B. Maximum Control

Author: **Alfonso[2] Peterssen**

B1. Easy

The problem is asking to count the number of leaves in a tree. To do this, just keep track of the number of edges (the degree) for each node. Then count the number of values equal to 1.

B2. Medium

To begin, note that it only makes sense to send ships to remote planets (leaves in the tree). And for $K = 2$, the answer is just the tree diameter (length of the longest path in the tree). One algorithm to find the diameter is based on BFS/DFS and works as follows: run a graph search from any vertex u and compute distances to every other vertex. Let the farthest vertex be v. Now run a graph search again from v and find the farthest vertex w. Then $v - -w$ is a diameter.

The general algorithm is also greedy and works as follows. First find the diameter. Then, for each $k = 3, 4, ..., n$, find the leaf that is the farthest from the set of already controlled planets, take it, and continue. We leave the proof of why this gives the optimum answer as an exercise. Instead, we explain how to implement this fast (in time $O(n \log n)$).

We can first find the above vertex v and root the vertex at it. For every vertex, compute its depth, i.e. the longest length of a path from it downward to a leaf. That is, the depth of a leaf is 0, and the depth of any other vertex is one plus the maximum depth among its children. We will maintain a priority queue of all vertices, with the key being their depth. Initialize it to hold all vertices except v. Then, for each $k = 2, 3, ..., n$, do the following:

- Find the highest-depth vertex w. (In the first iteration, it will the (only) child of v.)
- Go down from w to the farthest leaf. Remove all of them from the priority queue, and for each of them add 1 to the result.
- Print the current result.

(If at any point the queue becomes empty, then stop and just output the current score, which is n, as many times as necessary.)

The above can be also implemented in $O(n)$ time, using an array (of vectors) instead of a priority queue.

C. Encryption

Author: **Slobodan Mitrović**

C1. Easy

We compute prefix and suffix sums modulo p of the input array A. This is done in $O(N)$ time. Once we have these sums, we iterate over all the possible splitting points, $O(N)$ many of them, and for each compute the sum of the corresponding prefix and suffix sum. The maximum of those sums is the desired answer S.

C2. Medium

Notice that the problem does not change if we redefine $A(i) = A(i) \mod p$. Hence, without loss of generality assume that $0 \leq A(i) < p$, for all i. Let $A(i, j)$ denote the contiguous subarray of A beginning at the position i and ending at the position j. Let $g(i, j)$ define the score of $A(i, j)$. We solve this problem by dynamic

programming. To that end, define $dp(k, i)$ as the maximum score of splitting the prefix of A of length i into k pieces. The value $dp(k, i)$ can be obtained as follows:

$$dp(k, i) = \min_{j<i} dp(k-1, j) + g(j+1, i). \tag{1}$$

This solution takes $O(N^2 k)$ time, which is too slow.

Improved approach. Define $f(k, i, m)$ as

$$f(k, i, m) = \max_{g(j+1,i) \mod p = m} dp(k-1, j) + g(j+1, i).$$

Then, $dp(k, i)$ can be computed as follows

$$dp(k, i) = \max_{m} f(k, i, m).$$

For all $m \neq 0$, we can update $f(k, i, m)$ by using the following relation

$$f(k, i, m + A(i) \mod p) = f(k, i-1, m) - m + (m + A(i) \mod p).$$

Notice that the above equality is just a rotation by $A(i)$ of the values already computed. When m is equals 0, then we have also to consider the case in which the k-th group consists of only $A(i)$. Hence,

$$f(k, i, A(i)) = \min\{f(k, i-1, 0) + A(i), dp(k-1, i-1) + A(i)\}.$$

The running time of this approach is $O(Nkp)$, which passes within the time limit.

C3. Hard (feat. Jakub Tarnawski)

First, observe that this subtask asks for minimizing the total score, instead of maximizing as in the previous two versions. If $k \cdot p \geq N$, we can use the same approach as for the medium subtask (appropriately replacing maximization by minimization), which is fast enough to run within the time limit. In the rest of the explanation, we assume that $k \cdot p < N$.

Let x be the sum of the elements of A modulo p. Observe that S modulo p equals x, and hence we have that $S = x + jp$, for some $j \geq 0$.

Next, consider all the prefix sums of A. Then, as $k \cdot p > N$, by the pigeonhole principle there are at least k prefixes that have the same value modulo p. Let the first $k-1$ positions of those prefixes be $i_1 < i_2 < \ldots < i_{k-1}$. Consider the split of A into the following k parts: $A(0, i_1)$, $A(i_1 + 1, i_2)$, $\ldots$, $A(i_{k-2} + 1, i_{k-1}), A(i_{k-1} + 1, N-1)$. Then, the score of all the parts but the first and the last one will be 0. It is easy to see that the sum of the scores of the first and the last part will be either x or $p + x$. We now discuss how to check whether S equals x. If $S \neq x$, it will immediately imply that $S = x + p$.

Let $t_1, \ldots, t_k = N - 1$ be the positions of a splitting of A into k parts that achieves the minimum total score. Recall that $g(i, j)$ denotes the score of $A(i, j)$. Then, if $g(0, t_j) > g(0, t_{j+1})$, for any valid j, the value of S is larger than x. Otherwise, if $g(0, t_1) \leq g(0, t_2) \leq \ldots \leq g(0, t_k)$, the value of S equals $g(0, t_k) = x$. Hence, to test whether $S = x$ or not, it suffices to find the longest increasing sequence (LIS) on prefix sums such that the sequence ends at $g(0, N-1)$. If the length of LIS is at least k, S equals x, and otherwise S equals $x + p$. LIS can be constructed in $O(N \log N)$ time.

Another (slower but AC-able) solution to this problem is again dynamic programming, but with a cleverer way to compute (1). The key observation is that the term we want to minimize is separable in i and j (modulo p), i.e. it can be rewritten as two parts, one independent of i and the other independent of j. Therefore, for a fixed k, we compute the values $dp(k, i)$ in increasing order of i and, at the same time, maintain the minimum value of the part related only to j (modulo p). It turns out that the operation we need to consider is an update of a range with an arithmetic series, and thus can be done with a segment tree in $O(\log p)$ time for both update and query. The overall time complexity of this solution is $O(Nk \log p)$.

D. Hyperspace Jump

Author: **Jakub Tarnawski**

D1. Easy

The first step of the solution is to read the input and parse it, computing the value of the expression. One can treat it either as a floating-point number, or as a fraction (i.e. two numbers). In the latter case, it has to be reduced by dividing both the numerator and the denominator by their GCD (greatest common denominator).

To find the number of equal values for each value, there are many possible approaches. However, it has to be done fast – the naive approach runs in time $O(m^2)$, which is too slow.

One way is to use a data structure representing a dictionary that allows for operations in $O(\log m)$. We make two passes. In the first pass, for each value, we increment the corresponding counter in the dictionary. After the first pass, the dictionary will, for every value, hold the number of its occurrences. In the second pass, we can just print the answers using the dictionary.

If one used the fraction representation, then we can use just an array instead of a dictionary to do the above. Just count the number of times that the reduced fraction a/b appears in an array cell $t[a][b]$.

Another way is to sort the values. Then, to answer each question, we can (using binary search) find the lowest and highest index in the sorted array where the same value appears, and just subtract one from the other (and add 1) to obtain the answer. Many other similar approaches are possible.

D2. Hard

One can consider an $\Omega(m^2)$ solution, where we try to compare the subspaces in pairs. However, unless we come up with some extremely fast way of determining whether two subspaces are equal (much faster than, say, Gaussian elimination in $O(d^3)$), then this does not seem very hopeful.

So we will need to find some way to *normalize* a subspace. One such way is to compute its *projection matrix*, which does not depend on the choice of basis (or a generating set of vectors) for the subspace.

The first step in computing this matrix is to obtain an *orthonormal basis* for the subspace. The well-known way of doing so is called the Gram-Schmidt orthogonalization. It consists in, for every vector, projecting this vector onto the subspace spanned by the previous vectors. This process is well-described on Wikipedia. In this way, we obtain a basis that is orthonormal, that is, every vector in it has length 1 and every two different vectors are orthogonal (have inner product 0). (Note that this basis is unfortunately not unique for a subspace, so we cannot stop at that.)

Given an orthonormal basis $v_1, ..., v_k$, the projection matrix can be computed as:

$$P = \sum_{i=1}^{k} v_i v_i^\top$$

The name comes from the fact that for any vector $v \in \mathbb{R}^d$, the vector Pv is the projection of v onto the subspace. From this, it is easy to see that this matrix must be unique for a subspace.

Okay, so now we have a matrix for each subspace. However, it is has been computed with some small error that follows from floating-point computations. How do we now compare the subspaces fast?

One way is to compute a random hash of this matrix. For instance, sample random numbers $h_{11}, ..., h_{dd} \in [0, 1]$. Then, for each matrix, compute the number $\sum_{i,j} P_{ij} h_{ij}$. For two spaces that are equal, these numbers will be very close to each other (closer than, say, 10^{-12}). For two spaces that are distinct, these numbers will be different.

So we can sort these hashes (remembering which subspace each hash comes from) and process them in order. We form groups as we go. Whenever there is a sufficiently large gap, we end the current group and start a new one.

What is a sufficiently large gap, though? Well, if we assume that all subspaces are different (this would be the worst case), then we have $m \leq 30000$ numbers that lay randomly in some interval (at least $[0, 1]$, perhaps larger). A birthday-paradox type of phenomenon implies that the smallest distance between two of them is likely to be of the order of $O(1/m^2)$. Therefore choosing a gap of at most 10^{-10} should be good enough.

Instead of choosing a good value for this gap (which is, in the end, a bit of trial-and-error), one can instead perform some sort of two-phase bucketing: first group together all subspaces whose hashes are closer than some large threshold (say 10^{-6}) and then, in each group, rerun this process (with new random values of h_{ij}) to subdivide these preliminary groups into the final division.

There is also an alternative solution, where we pick a random point in the space and compute its distance to each subspace. With probability one, for different subspaces this distance is different. However, doing so also involves computing the projection matrix. We also need some way of comparing the m numbers, which is also the same as above. Therefore this solution is more or less equivalent to the first one.

Another solution is to perform Gaussian elimination in a certain canonical way (for example consider having the first nonzero entry in each row of the resulting upper-triangular matrix be equal to 1).

Finally, it is possible to implement the above solutions in a finite field (i.e. in integer arithmetic modulo some prime number). It is then useful to use several primes in parallel to obtain a good enough accuracy (that follows from Chinese remaindering).

E. Guard Duty

Author: **Chia-An Yu**

E1. Easy

First of all, the answer is no if $R \neq B$ because there is no perfect matching. To check if there is a perfect matching satisfying the conditions, it is possible to go through all the assignments and check if any of them is good. There are $O(R!)$ assignments and checking each one of them takes $O(R^2)$. This gives the naive solution and is enough to pass all the tests, but rather difficult to implement.

A more elegant $O(1)$ solution is to use the fact that the answer is yes if and only if the number of spaceships and bases are the same. In fact, an assignment whose sum of segment (path) lengths is minimum will not have any intersection. We can prove it by contradiction. Let's assume that there exists a pair of intersected segments $\overline{AB}$, $\overline{CD}$, where A and C are spaceships and B, D are bases. If we change the assignment to $\overline{AD}$ and $\overline{CB}$, due to the triangle inequality, the sum of segment lengths will decrease and we obtain a new assignment whose sum of segment lengths is smaller than the minimum value. A contradiction. Therefore, it is guaranteed to have an assignment without intersections as long as the number of spaceships and bases are the same.

TL;DR: Just answer YES if $R = B$ and NO otherwise.

E2. Medium

We sort the times in increasing order and consider the differences between neighboring points. Then, the problem becomes computing the minimum of taking K elements from a series $d_1, d_2, \ldots, d_{N-1}$ with the constraint of no two taken elements are adjacent.

This can be solved by an $O(KN)$ DP. Let $DP(i, j, used)$ denote the minimum value we can get by taking j elements from the first i elements with $used$ indicating the ith element being taken or not. The transformation formula is

$$DP(i, j, unused) = \min(DP(i-1, j, used), DP(i-1, j, unused)$$
$$DP(i, j, used) = DP(i-1, j-1, unused) + d_i\,.$$

However, this is not fast enough. We use the observation that d_i will not be taken if it is not one of the $3K$ smallest element. Indeed, if such d_i is taken, we can always replace it with one of the $3K$ smallest element while satisfying the constraint because at most $3K-3$ of them is occupied. Therefore, we only need to consider at most $3K$ of the elements. This gives us a solution with complexity $O(K^2)$ and is enough to pass all tests. Note that there are faster solutions but it is not required for this problem.

E3. Hard

There are many ways to solve this problem. The solution discussed here is to reduce the problem into subproblems so that their solutions are independent. Let's call a set of points balanced if it has the same number of spaceships and bases. From the solution of the easy problem, we know that every balanced set has a solution. Therefore, if we can find a line that separates the set into two balanced subsets, we can solve these two subproblems separately and then combine the results.

So how do we find such a line? Let's assume that the leftmost point is a spaceship. We first shift it to the origin and then sort the other $2N-1$ points by angle. We scan through the points and keep track of the value

$$\#\text{ spaceships we have passed } - \#\text{ bases we have passed}\,.$$

The value starts with 0 and ends with -1 because there is one more base than there are spaceships. If we look at the point where the value first becomes -1, we see that it will be a base and the points before and after both form a balanced set. This algorithm takes $O(N \log N)$ time to reduce the problem and the subproblem has size at most $N-1$, so the overall time complexity is $O(N^2 \log N)$.

This algorithm may not fit in the time limit depending on the implementaion. Nevertheless, some optimizations can be done, such as applying a random rotation before reducing the problem, and partitioning the set so that the difference in size of the two subsets is minimized. These do not improve the complexity in the worst case, but they are nevertheless very effective.

F. Lightsabers

Author: **Damian Straszak**

F1. Easy

This is a straightforward problem and high efficiency of a solution was not really required to fit in the time limit. Let us anyway describe a solution that runs in optimal, linear time $O(n)$.

We slide a window of length $k = \sum_{i=1}^{m} k_i$ over the sequence of knights and for every color i we maintain the number of lightsabers of this color in the currently considered interval. When we move such a window by one position to the right we just need to update two counts (for the knight who left the window and the one who entered) – this is done in $O(1)$ time. We also need to check if this particular window has the right counts of colors. Doing it naively would cost $O(m)$ time (check every color separately). Instead we can also maintain another count C: the number of colors i for which the current count c_i is equal or exceeds the desired count k_i, i.e., for which $c_i \geq k_i$. Then, the current window has the correct counts if and only if $C = m$. It remains to note that C can be also updated in $O(1)$ time every time the window is shifted by one position. Thus the total running time is $O(n)$.

F2. Medium

This subproblem turns out to be also solvable in time $O(n)$ and the solution is a simple extension of the idea presented above for the Easy variant.

For every position $j = 1, 2, \ldots, n$ in the sequence we would like to find the smallest possible index j' such that if $(c_1, c_2, \ldots, c_m)$ is the sequence of counts of the subsequent colors $1, 2, \ldots, m$ in the interval from the jth to the j'th knight then $c_1 \geq k_1$, $c_2 \geq k_2$, $\ldots$, $c_m \geq k_m$. This means that we need to remove $\sum_{i=1}^{m}(c_i - k_i)$

knights from this interval to achieve our goal. If we compute this number for every starting position j then their minimum is the solution to the problem!

The remaining question is how do we compute the index j' and the corresponding counts $c_1, c_2, \ldots, c_m$ efficiently? We can use a similar "two pointers" sliding algorithm to do that: whenever we move j one position to the right, the corresponding j' will also either move to the right (by one or more positions) or stay the same. By maintaining the counts in the current interval we can move any of these pointers in $O(1)$ time. Since they will be moved in total at most $O(n)$ times, this is our total running time.

F3. Hard

This problem is of a slightly different flavor as the Easy and Medium variants and asks us to count the number of all possible groups of knights with different lightsaber color patterns.

Perhaps the simplest way to approach this problem is via dynamic programming. We can consider all knights with lightsaber colors $1, 2, \ldots, l$ and denote by $dp[l, j]$ the number of possible color patterns for groups of $0 \leq j \leq k$ knights among them. Then we can compute every number $dp[l, j]$ in roughly $O(min(j, c_l))$ time, where c_l is the number of knights with a lightsaber of color j, or even in $O(1)$ using prefix sums of $dp[l-1, \cdot]$ values. Nevertheless it is not possible to beat $O(mk)$ running time using this approach, which is definitely too slow.

To the rescue comes an interpretation of this problem using polynomials. For every color $l \in \{1, 2, \ldots, m\}$ define a polynomial

$$p_l(x) = \sum_{j=0}^{c_l} x^j.$$

Then, it is not hard to see that the coefficient of x^k in the polynomial

$$p(x) = \prod_{l=1}^{m} p_l(x),$$

is exactly the number we are looking for. However, computing the product of these polynomials in the standard way can be seen to be equivalent to the dynamical programming algorithm described above! To obtain a speed-up we apply the FFT algorithm to multiply these polynomials more efficiently.

More precisely, FFT allows us to compute a product of two polynomials of degree at most d in $O(d \log d)$ time. Since there are more than two polynomials to multiply here, we can try to pair them up so as to always multiply two polynomials of roughly the same degree. This leads to an algorithm with $O(n \log^2 n)$ running time.