Editorial

Tasks, test data and solutions were prepared by: Adrian Beker, Marin Kišić, Daniel Paleka, Ivan Paljak, Tonko Sabolčec, Bojan Štetić i Paula Vidas. Implementation examples are given in attached source code files.

Task: Semafor

Prepared by: Bojan Štetić, Ivan Paljak, Daniel Paleka
Necessary skills: matrix multiplication, graph theory, binary exponentiation

We will solve the harder case M=2.

We represent a state of the board by a bitmask of 10 bytes. Call a bitmask *nice* if the represented board shows a valid number.

It is well known that the vertices of the *hypercube graph* are all bitmasks of some size (10 here), and the edges connect bitmasks which differ in exactly one place. Let H denote the adjacency matrix of the hypercube.

Then the problem, formally, asks for the following: for each nice bitmask, find the number of walks of length N in the hypercube, starting from the state X, such that the walk visits the set of nice bitmasks every K steps.

Claim 1: The number of walks of length D in the hypercube, starting in X and ending in Y, equals the (X, Y)-th entry in the matrix H^D.

Proof: Look into Daniel A. Spielman's "Spectral and Algebraic Graph Theory", brilliant stuff. For an elementary proof, it's Lemma 3 on the link:
https://courses.grainger.illinois.edu/cs598cci/sp2020/LectureNotes/lecture1.pdf □

Definition 2: Let B be the principal submatrix of H^K indexed by only the rows and columns of the nice bitmasks.

Claim 3: The number of walks (from X to Y) of length N in the hypercube, where K divides N, such that every K steps the walk visits a nice bitmask, equals the (X, Y)-th entry in the matrix $B^{N/K}$.

Proof: Left as exercise. □

There are several steps in the solution. We need to calculate the matrix B, then calculate $B^{N/K}$, and then multiply it with the matrix corresponding to the last $N \mod K$ steps. We omit the third because it is very similar to the first step, so we assume K divides N in the rest of the exposition.

Let us solve the second step (raising B to the N/K-th power) first. It is enough to use naive binary exponentiation and multiply two matrices in the ordinary cubic complexity, because for $M = 2$ we are dealing with 100×100 matrices.

The first step is tougher, because B is a submatrix of H^K, and H is a 1024×1024 matrix for $M = 2$. Naive calculation of H^K passes only the subtask $M = 1$. For smaller K and $M = 2$, it can be done with clever dynamic programming, and this should give the first four subtasks.

For the full problem, we need to raise the hypercube matrix to some power faster.

Claim 4: The (X, Y)-th entry of the matrix B depends only on the number of bits in which the bitmasks X and Y differ.

Proof: Without losing of generality, we can XOR all vertices of the hypercube with some fixed bitmask. Thus, we can assume $X = 0$. It's also clear that the order of the bits doesn't matter at all. Hence, the number of walks from 0 to Y depends only on the bitcount of Y.

Claim 4 shows that's it's enough to calculate the first row of the matrix H^K. Depending on the exact implementation, it can pass the first five subtasks, or even pass the whole problem. (Ask Dorijan Lendvaj for the XOR-convolution solution of quadratic complexity that runs faster than the official solution.)

The official solution calculates the numbers of walks starting in $X = 0$ in a smarter way. It's enough to calculate, for every $0 \leq r \leq 10$, the number of walks starting from bitcount 0 and ending in bitcount r.

The number of ways in which we can get from one bitcount $0 \leq r \leq 10$ to another is given by the following 11×11 matrix:

$$C = \begin{bmatrix} 0 & 10 & 0 & 0 & \dots & 0 & 0 & 0 \\ 1 & 0 & 9 & 0 & \dots & 0 & 0 & 0 \\ 0 & 2 & 0 & 8 & \dots & 0 & 0 & 0 \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ 0 & 0 & 0 & 0 & \dots & 9 & 0 & 1 \\ 0 & 0 & 0 & 0 & \dots & 0 & 10 & 0 \end{bmatrix}$$

By raising the matrix C to a suitable power, we can get the number of walks of length K from bitcount 0 to each bitcount $0 \leq r \leq 10$. To get the actual number of walks in the hypercube, we just need to divide by a suitable binomial coefficient, due to symmetry. For implementation details, see the official source codes.

There is an even faster solution, which uses *exponential generating functions*. The number of walks of length K from the bitmask 0 to some mask with $10 - r$ bits equals the coefficient next to $\frac{x^K}{K!}$ of the following expression:

$$\left(\sum_{i=0}^{\infty} \frac{x^{2i}}{(2i)!}\right)^r \left(\sum_{i=0}^{\infty} \frac{x^{2i+1}}{(2i+1)!}\right)^{10-r} = \left(\frac{e^x + e^{-x}}{2}\right)^r \left(\frac{e^x - e^{-x}}{2}\right)^{10-r}.$$

By expanding the expression carefully, we get a solution of the complexity $O((5 \cdot M)^2 + (5 \cdot M) \log K)$, that is, entries of the $n \times n$ hypercube matrix raised to some power K is possible in $O(\log n \log K)$ time.

For similar ideas, look into Herbert Wilf's free book *generatingfunctionology*.

Task: Zagrade

Prepared by: Paula Vidas
Necessary skills: stack, parentheses, ad hoc

We first describe the solution for the case when the whole password is a valid sequence. In the first subtask we have $Q = \frac{N^2}{4}$, so we can query every even lenght interval (note that it makes no sense to query odd lenght intervals). One possible solution is the following: we ask for the first two charactes, first four, and so on unitl we get a positive answer. We then know the position of the close parenthesis which is paired up with the open parenthesis on the first position. Now we can recursively solve the same task for the interval between these parentheses, and for the interval that is on the right of the close parenthesis.

In the third subtask, we can ask $Q = N - 1$ queries. We will use a stack, which is empty at the beginning. We traverse the positions in order. If the stack is empty, push the current position on the stack. Otherwise, ask the query for the interval between the position on top and the current position. If the answer is positive, those parentheses are paired up (left is open and right is close), and we pop the stack. If the answer is negative, we push the current position on the stack. In the end, the stack will be empty.

What if the whole password is not a valid sequence? The algorithm is the same, but the stack doesn't have to be empty in the end. First half of the remaining postions must have a close parenthesis, and the second half an open parenthesis.