

Problem A. Best Waifu

All you need to ac this problem is trust and good taste.

Problem B. Sanity

This is just a simple implementation problem: we keep a count of the amount of days since Chessbot last emailed Bossologist, and every K 'th day we increase his sanity by one. You can do this through modulus or resetting the count every K days.

Problem C. Calendar

Let's define an array v where $v_i = 1$ if there is an event on the i th day of the year. The essentially asks for the longest chain of 0's in v . This can be done by keeping a counter of the current chain of 0's. Loop i from 1 to 365. If $v[i] = 0$ you can increase the counter and update the answer. However, if $v[i] = 1$ the counter must be reset to 0 (note that the answer is not reset).

In order to generate the array v , a while loop can be used to start at a day and increment the day by C until it passes 365. The number of days in each month is [31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31] as 2023 is not a leap year.

The final complexity is $O(365 \cdot N)$.

Problem D. Penalty

Say the negative number is at position x, y . Also let $f(a, b, c, d) = \sum_{i=a}^b \sum_{j=c}^d v[i][j]$. The answer is

$$\max(f(1, n, 1, m), f(1, x-1, 1, m), f(x+1, n, 1, m), f(1, n, 1, y-1), f(1, n, y+1, m))$$

Problem E. Mixed Economy

To solve this problem we have to do two things:

1. Figure out how much each person has in their bank account before the tax
2. Find the person that needs to pay the tax and figure out how much they need to pay

The first task is simple enough, you can iterate through the spending list and keep track of each person's bank account using a map or an equivalent data structure. For the second task, we will iterate through the list and keep track of two variables: a and b . a is the length of the longest spending streak by any person so far, and b is the length of the current spending streak. Whenever the spending list switches to another person, we reset b to 1. Every time b surpasses a , we set a to b and label the person we are currently iterating through as the one that pays the tax.

After doing this we simply subtract $a \cdot (N-1)$ dollars from the taxed person and give a dollars to everyone else. To get the answer just find the person with the least amount of money and subtract that value from the person with the most money.

Problem F. Cloning

There are many more complicated solutions to this problem, but I present a simple one.

Since the answer has to have the same number of clones and trees, the length must be divisible by two (even). So the sufficient condition for there to be an answer is that none of the intervals cover an odd length. Otherwise, there always exists a solution that is "MTMTMT ... ". Since in some interval, the number of even indices and odd indices must be the same, and hence the number of clones and trees must also be the same.

The final complexity is $O(n + m)$ per testcase (to take in input). Due to the fact that the problem is multitest, it is important to not return right while taking in input, or else stray input might mess up future test cases.

Problem H. Zero Trust

The question is essentially asking the number of students with positive trust as well as the sum of all the students with positive trust. Let $d_i = \sum_{j=0}^i x_j$. If a student a has positive trust, their contribution to the final sum is $t_a - d_i$. It is important to note that the d_i term is independent of the student. So if the number of students with positive trust is s , they will all contribute $-s \cdot d_i$ in total. So it suffices to only maintain the sum of t_a for all the students who still have positive trust. The intended solution uses a min priority queue, removing students and decreasing the total sum of t each time the minimal trust student a has a smaller (not strictly) value of t_a than d_i .

The final complexity is $O(n \log n + q)$. Note that many other solutions such as sorting or prefix sums also work as well.

Problem I. Counting Flags

the editorial has been written on overleaf for better visuals and explanatory code.

<https://www.overleaf.com/project/62e4124187cdae1c10a99ffd>

Problem J. Thomas Game Revisited Again

The most important observation of this problem is that the contribution of each character is independent of the other ones. We will compute the answers for the numbers $1, 2, \dots$ and the answers for the "non-numbers" separately.

With the numbers removed, each line looks as follows

$c[i] = a[i] * b[i] + a[i] * b[i] \dots$;

There are exactly N "clumps" of these $a * b$ terms.

So the contribution table is as follows for line:

- 1 occurrence of c
- n occurrences of a
- n occurrences of b
- $n - 1$ occurrences of $+$
- n occurrences of $*$
- 1 occurrence of $=$
- 1 occurrence of $;$
- $4 \cdot n + 2$ occurrences of $[$
- $4 \cdot n + 2$ occurrences of $]$

There are n^2 lines, and this gives us an $O(1)$ way to find the contribution of those characters.

Now for the contribution of the numbers. There is a much for formal and bash heavy way, but I present a simpler method. So by the symmetry of matrix multiplication, it makes sense to think that each number occurs the same number of times in the final expanded code. There are a total of $n^2 \cdot (2 + 4 \cdot n)$ places for numbers to appear in the final expanded code. So if we divide by n , we can get the amount of times each number individually occurs. So let $x = n \cdot (2 + 4 \cdot n)$. So for each number i from $1 \dots n$, we need to count the $(\# \text{ of digits in } i) \cdot x$ and sum it up. We can for each possible $(\# \text{ of digits in } i)$ we count the number of $i \leq N$ that have that number of digits. So let p_i be the number of numbers that have exactly i digits (without leading 0's).

$$p_i = \min(N, 10^i) - \sum_{j=0}^{i-1} p_j + [i = 0]$$

The final term is just means 1 if $i = 0$ and 0 otherwise. This term is needed to count the contribution of the value 0 itself. This gives us an $O(\log N)$ way to evaluate the contribution of the numbers.

The final complexity is $O(\log N)$ per testcase. It is also worth noting that that answer itself is not very large, ie only of the order of N^3 or so, but since $N \leq 10^9$, finding the answer under mod is still easier than outputting something of the order of 10^{27} .

Problem K. Rock Paper Scissors (Easy Version)

The model solution uses too much math + is really painful to explain + references a math.stackexchange post so we will present a tester solution that uses most of the same logic/complexity written by hyforces.

Let dp_x represent the expected number of games for one Misaka clone to emerge victorious with x Misaka clones initially. Obviously, $dp_1 = 0$. Let us try to find an equation for this quantity.

The probability that a single game results in the elimination of some Misaka clones with x clones at the beginning of the game can be expressed as $3(\frac{2}{3})^x - 6(\frac{1}{3})^x$. For some clones to be eliminated, all the clones must pick between two choices, as if all three choices are present then no eliminations will occur. Then, we must subtract the chance that all clones pick one of two options. If we fix the two options that the clones will pick between, the chance is $(\frac{2}{3})^x - 2(\frac{1}{3})^x$. Multiply by 3 to account for all the possible pairs of choices of rock-paper-scissors to get the final probability of $3((\frac{2}{3})^x - 2(\frac{1}{3})^x)$. Therefore, the expected amount of games before a clone is eliminated is simply the reciprocal of that quantity, or $\frac{1}{3(\frac{2}{3})^x - 6(\frac{1}{3})^x}$.

In the case at least one clone is eliminated, dp_x can transition to any one of $dp_1, \dots, dp_{x-1}$ depending on how many clones are eliminated. The chance that k clones are left alive following a match follows the binomial distribution, and since there are $2^x - 2$ possible ways for each of the clones to pick between our two options (minus the configurations where all of the clones pick one option), the chance that k clones are left alive after a game where it is guaranteed at least one clone is eliminated is $\frac{\binom{x}{k}}{2^x - 2}$: we pick which k clones pick the option that does not get them eliminated, and then divide by the total amount of configurations. Therefore, combining these two facts gives us the final dp equation:

$$dp_x = \frac{1}{3(\frac{2}{3})^x - 6(\frac{1}{3})^x} + \frac{\sum_{k=1}^{x-1} dp_k \binom{x}{k}}{2^x - 2}$$

The difference between the model and hyforce's solution is that the contribution of the current state before it goes to the smaller states is computed differently.

This dp formula can be evaluated in $O(n)$ time for each i , giving us an $O(n^2)$ solution in total. As an implementation detail, using inverse mod to compute binomial coefficients each time adds an extra factor of log, so using precomputed factorials and inverse factorials is recommended instead.

Problem L. Quantum Schmovements

There are two critical observations needed to arrive at the full solution.

First, any move can be undone or there is no danger of going to a node and not being able to come back. Consider some edge that must be cross from u to v with label l . Assume WLOG that Waymo is trying to cross from u to v while Thoams is standing on node l . After Waymo crosses from u to v , Thomas is still standing on node l so Waymo can come back to node u immediately if needed (or continue moving). This turns the problem from a maximization problem to simply counting the number of nodes that can be reached.

The second important observation is that the number of states (t, w) where Thomas is on node t and Waymo is on node w which can be reached from the initial state is very limited. Consider a state (w, t) . In order to have reached this state from some other state an edge was crossed. To make it possible to cross an edge, one person must have been at the endpoint of an edge while the other person was at the label of the edge. This means the number of reachable states cannot be much larger than $2 * m + 1$.

Combining these two facts together, we can come to a solution. Since the number of possible states is small, it is possible to just dfs over all of them. By maintaining for every state (u, v) what w exists such that (u, v) can reach (w, v) , we are able to do a dfs. Such an edge list can be built as follows. Consider an edge from u to v with label l . This edge from (u, l) to (v, l) exists and vice versa. This adjacency list can be an array of maps or hashmaps to avoid mle. The final complexity is $O(M \log M)$ assuming a regular map was used.

Problem M. Driving

So for starters. -1 is only possible as an output if no two cool cities are connected. This can be checked separately with a DSU or another data structure, so we will only focus on the case when at least one pair of cool cities are connected.

This problem is essentially asking for the "minimum maximum" of any path between any pair of cool cities. After adding a new cool city, we just have to check its minimum maximum to any other cool city and update the already known answer.

Whenever "minimum maximum" is seen, the first thought should be to binary search. Also whenever "connectivity of all edges with weights $\leq$ some value the first thought should be kruskal tree (reachability tree/line tree). In fact, there exists a way to binary search on a tree; it is called binary lifting. Also to check whether or not a given node in the kruskal tree/some weight of an edge, we just have to check if its subtree in the kruskal tree has more than one leaf node activated. This can be done with a simple node add, subtree sum with the help of a BIT and preorder traversal. The total complexity is $O(\log^2 n)$ per query. Personally, I found that boring but passable. However, the model solution is better, using $O(N)$ time over all Q queries at most.

The most important observation from the above solution is that we only ever "insert" or activate nodes. We never "delete" or deactivate nodes. So we can actually do a brute force. We start at a leaf node and go to its parent, marking it as visited. Then we go to its parent and so on until we reach some root node with no parent. Also, the moment we see a visited node (which we did not mark) we can stop immediately. This is because some other node has also marked this node already, so we know that this node's subtree has a sum > 1 . This complexity is actually amortized $O(N)$ since every node is only marked as visited once. This is a much better solution in my opinion, and I challenge you to think about what this algorithm means intuitively with your understanding of a kruskal tree. The final complexity is $O(M \log M + M \cdot \alpha(M) + N)$ since we have to sort the edges for kruskal as well.

Problem N. Schemy

Let's define a set of rocks on the grid as a configuration. A configuration is "valid" if there are no alternate paths Codicon can take and "invalid" otherwise. Additionally, a configuration is "optimal" if it is valid and has the least number of rocks needed to be valid.

First note that the desired path divides the grid into at most 2 sections which we'll call "top-right" and "bottom-left". We can solve each section separately, so let's focus on the top-right one since by symmetry it's the same as solving the bottom-left one.

Looking at the top-right section, we know that a configuration is invalid if there is a way for Codicon to divert from the path to the right (moving right instead of down), and then rejoining the path by moving down into it. So there are two ways for rocks to prevent an alternate valid path: block the ability of Codicon to move right, or block any "potential paths" from rejoining onto the main path. The most straight forward way to do this is either have rocks hug the right of the path, preventing Codicon from diverting from the path in the first place, or hug the top of the path to stop Codicon from rejoining onto

the path if he diverts from it. But can we always get an optimal configuration by only hugging the path? Well it actually turns out we can! The proof is pretty long and complicated so if you would like to see it check out <https://www.overleaf.com/read/grpjrpbbrxpkt>, otherwise you can just assume that this is true.

So all we have to do now is find the most optimal configuration only using rocks that are adjacent to the path. Recall that earlier we mentioned two possible ways to prevent an alternate path: stop Codicon from moving right instead of down, or prevent him from rejoining the path from above. So we don't have to worry about stopping him from rejoining the path until we let him divert from it. And similarly once there's no way for him to rejoin the path we can let Codicon divert from the path whenever he wants. Note however that as soon as we let him go on an alternate path, we must prevent him from rejoining in every step afterwards, which effectively means covering the top of the path with rocks. So basically the only configurations we need to look at are the ones where we first stop Codicon from diverting right by hugging the right side of the path with rocks, and then we switch to preventing him from rejoining the path which means covering the top of the path with rocks. We can just iterate through each of these configurations in $O(N)$ time, and the one that uses the fewest rocks is our answer.

Problem O. Highways

It is possible to go from a, b to c, d (assume $d > b$) iff for all x between a and c , there is a road with y coordinate $geqb$ that contains it. This motivates the following solution:

The idea is to process queries and roads with a sweep in ascending y . Maintain a lazy segment tree over the x coordinates. The underlying array $a[i]$ of the seg tree is the maximum y coordinate of a road such that i is an x coordinate covered by it.

Then as you iterate over y , each road is added to the segtree via a range set update (set all $a[x] = y$ over range of the road).

Then, to answer a query, we query the minimum of the range a to c in the segtree. If this is $\geq b$, then we have a valid way to reach it.

Here is the link to a tester solution: <https://pastebin.com/HtKcv1Ea>.

Problem P. Waku-Waku Abyss



Brute Force (Tests 1-2):

Clearly we cannot enumerate all paths, so we must use dynamic programming. The brute force dp approach is as follows. Let $DP[j]$ be the shortest path from city 1 to city j and $c_{i,j} = (A_{i+1} \oplus A_{i+2} \oplus \dots \oplus A_j) - 16$ be the cost of the path p from city i to city j (assuming p exists of course). Then our transition is:

$$DP[j] = \min(DP[i] + c_{i,j})$$

We will iterate over all M regions in order and perform the corresponding dp transitions. There are M regions each with up to $O(N^2)$ edges for a total of $O(M \cdot N^2)$ edges/transitions.

Time Complexity: $O(M \cdot N^2)$

Full Solution:

Reinterpretation of DP:

Clearly, we need to optimize our dp. The key insight is that $c_{i,j}$ can only take on $\max(A_i) + 1 = 2^5$ values. Thus, we can rewrite the transition as:

$$DP[j] = \min_c (c + \min(DP[i]) \mid c_{i,j} = c)$$

In other words, we iterate over all 2^5 distinct values of $c_{i,j}$ and for each value c we find the minimum $DP[i]$ such that the cost of going from i to j is c . This reinterpretation of the dp transition is insightful because we've removed $c_{i,j}$ from the original min function. It may now be possible to compute $\min(DP[i])$ quickly even when transitioning from j to $j + 1$.

Insight + Motivation:

How do we compute $\min(DP[i])$ quickly? Notice that if $c_{i_1,j} = c_{i_2,j}$, then $c_{i_1,j+1} = c_{i_2,j+1}$ (assuming the corresponding paths exist of course). Thus, the set of i considered in $\min(DP[i])$ for a given j will be considered again in $\min(DP[i])$ for every $j_2 > j$ (assuming $j_2 - i \leq L_k$). This stability in the set of i being considered means we should try to maintain it as we iterate over j .

Conceptual Procedure:

Denote $C[x]$ as the set of $DP[i]$ such that the xor part of $c_{i,j} = (A_{i+1} \oplus A_{i+2} \oplus \dots \oplus A_j) = x$. After, when incrementing j to $j + 1$, we simply do $C[x] = C[x \oplus A_{j+1}]$ for all x . Then, we add $DP[j]$ to $C[A_{j+1}]$. Next, due to the L_k constraint, we remove $DP[j - L]$ from $C[A_{j-L+1} \oplus \dots \oplus A_{j+1}]$. Finally, we need to perform our actual dp transition, so we need to query $\min(C[x])$ for all x . Thus, there are a total of $O(\max(A_i))$ reassignments + 1 insertion, 1 deletion, and $O(\max(A_i))$ minimum queries on $C[x]$.

Formalizing Datastructure:

Each reassignment can be done in $O(1)$ if we use references or a reinterpretation of the indices that is equivalent. Finally, we don't want $C[x]$ to be a set, but rather a datastructure that supports quick insertion, deletion, and minimum. A set or priority queue works in $O(\log(N))$ per operation. However, the queue like insertion and deletion motivates us to use a min queue for $O(1)$ per operation. Either solution will be fast enough.

Time Complexity: $O(M \cdot N \cdot \max(A_i))$ or $O(M \cdot N \cdot \max(A_i) \cdot \log(N))$

Problem Q. Food Poisoning

So let's define P as the set of primes as defined in the problem. Let Q be all the primes from $1 \dots M$ that are not in the set. The number of elements divisible by at least one prime in P is the same as $N -$ all the numbers which only have prime divisors in Q . So now we define S as all the numbers which only have divisors in Q (note that 1 is in S).

Let v_i be the number of elements x from $1 \dots i$ such that x is in S . We want to maintain v_n . We will focus on insertion operations for now since deletion is similar. The motivation for the solution is that when a prime p is inserted, v_x for some x will increase by $v_{\lfloor \frac{x}{p} \rfloor}$. The intuition is that for some number i in S such that $1 \dots \lfloor \frac{x}{p} \rfloor$, $i \cdot p$ is also in S . Since we want to maintain v_n , we will need to know the v_i for some i such that $i = \lfloor \frac{n}{j} \rfloor$ for some number j . We also need v_x for all x such that $x = v_{\lfloor \frac{n}{j} \rfloor}$. We now insert a proof

by hyforces on why $\lfloor \frac{\lfloor \frac{n}{j} \rfloor}{j} \rfloor = \lfloor \frac{n}{ij} \rfloor$

If we have $\lfloor \frac{x}{ab} \rfloor = k$, we know that $x = k(ab) + l$ for some l in the range $[0, ab)$ and integer k . Evaluate $\lfloor \frac{x}{a} \rfloor = \lfloor \frac{k(ab) + l}{a} \rfloor = \lfloor kb + \frac{l}{a} \rfloor$. Since kb is an integer, we can extract it out of the floor operator: $kb + \lfloor \frac{l}{a} \rfloor$. Dividing by b and taking the floor, which will give us our entire expression $\lfloor \frac{\lfloor \frac{x}{a} \rfloor}{b} \rfloor$, we get the following expression: $\lfloor \frac{kb + \lfloor \frac{l}{a} \rfloor}{b} \rfloor = \lfloor k + \frac{\lfloor \frac{l}{a} \rfloor}{b} \rfloor$. Since k is an integer, we can pull it out to obtain $k + \lfloor \frac{\lfloor \frac{l}{a} \rfloor}{b} \rfloor$. Since

$l < ab$, we know that $\frac{\lfloor \frac{l}{a} \rfloor}{b} < \frac{l}{ab} = \frac{l}{ab} < 1$, so the expression evaluates to k in the end, as the term in the right floor is less than 1, proving our claim as desired.

Deletion is similar to addition, but instead of adding you are subtracting the contribution of $v_{\lfloor \frac{x}{p} \rfloor}$

There is also an important fact that for some integer N , there are only $O(\sqrt{N})$ unique values of $\lfloor \frac{N}{x} \rfloor$. The proof for this left an exercise for the reader.

This gives us an $O(Q\sqrt{N} \log N)$ solution where for each addition, you insert all powers of the primes p^i such that $p^i \leq N$. Similarly, for deletion, we also delete all powers of primes.

The way to get rid of the log is to not consider each prime power separately, but rather let them stack. When inserting, we increase each v_x by $v_{\lfloor \frac{x}{p} \rfloor}$ in increasing order of x . On the contrary, when we delete we decrease v_p by $v_{\lfloor \frac{x}{p} \rfloor}$ in decreasing order.

The final complexity is $O(\frac{N^{1.5}}{\log N} + Q\sqrt{N})$. The first part of the complexity is what is needed to insert all the primes initially. The second part is to answer each query in $O(\sqrt{N})$ time.

Problem R. Rock Paper Scissors (Hard Version)

I will assume you have solved the easy version before reading this. I don't know what dp formula you used, but it should have the form of

$$f_i = c_i + b_i \cdot a_i$$

where $a_i = \sum_{j=1}^{i-1} \binom{i}{j} \cdot f_j$ and b_i and c_i are values only based on i and independent of the dp.

The choose function is annoying to deal with, so we can expand $\binom{i}{j}$ to be $i! \cdot \frac{1}{j!} \cdot 1(i-j)!$. We can multiply b_i by $i!$ and redefine a_i :

$$a_i = \sum_{j=1}^{i-1} \frac{1}{(i-j)!} \cdot \frac{f_j}{j!}$$

This looks suspiciously like convolution, in fact, it is convolution!

The problem is that f has to be evaluated in order so that the future f values are correct. But this gives us a very important fact, we can compute the contribution of $f_{a...b}$ to $a_{c...d}$ where $a \leq b < c \leq d$ with a convolution. This also gives the motivation for the intended solution.

So say we wanted to solve $f_{[l,r]}$ and the contribution from $f_{[i,l]}$ to $a_{[l,r]}$ is all computed already. If $l+1 = r$, we can set $f_i = c_i + b_i * a_i$ and return right away. Otherwise we can define $m = \frac{l+r}{2}$. We can safely solve for $f_{[l,m]}$ since the contribution from $f_{[1,l]}$ is still already computed. Now we need to also count the contribution of $f_{[l,m]}$ to $a_{[m,r]}$ before solving $f_{[m,r]}$. We know that this contribution can be computed with a convolution as stated above.

So the final solution is just to use a divide and conquer. The merge function is $O(n \log n)$ and this gives us a total complexity of $O(n \log^2 n)$. A small detail is that you need to use fft under arbitrary mod to do the convolution, for that you can refer to kactl or ecnerwala's github for that.