Problem A
3 Partition

Author: Shreyan Ray

Let $S = \sum_{i=1}^{3N} A_i$. If S is not divisible by N, the answer is trivially `No`, because you need to form N triplets of equal sum.

Now, let $T = \frac{S}{N}$. We do casework on the value of T. Note that it can only be between 3 and 9. Each triplet we choose must have a sum of exactly T.

- $T = 3, 9$: It means that all elements are 1 or 3, respectively. Valid division is thus always possible.
- $T = 4, 8$: There is only 1 valid triplet with these possible sums, $[1, 1, 2]$ and $[2, 3, 3]$ respectively.
- $T = 5, 6, 7$: Each of these cases has exactly 2 valid triplets:
  - $T = 5$: $[1, 1, 3]$ or $[1, 2, 2]$
  - $T = 6$: $[1, 2, 3]$ or $[2, 2, 2]$
  - $T = 7$: $[1, 3, 3]$ or $[2, 2, 3]$

We see that in all cases there are at most 2 valid triplets. We can simply iterate on the count of the first type, and take $n - i$ of the second type; then see if the counts match the given count.

Solve Count: 202
First Solve: 00:13 by CodeDawgs

Problem B
Cache Optimization

Author: Shreyan Ray

We try to analyze when an element needs 2 cycles, and when it needs 1. Let L_i denote the last occurrence of A_i. If, it does not exist, obviously this will always take 2 CPU cycles.

Now, at L_i, A_i would have been inserted into the cache as the most recent element. If between L_i and i, the element got removed, then we need to spend 2 CPU cycles on A_i, otherwise only 1.

The condition for removal is pretty simple though. Note that since A_i was most recent at L_i, only things between L_i and i can be more recent. If A_i is no longer in the cache, it means at least K things have occurred between L_i and i, and the converse is also true.

Thus, let X denote the number of distinct elements between L_i and i. If $X \geq K$, then A_i goes out of cache and we need 2 CPU cycles, otherwise we need 1 CPU cycles. If we can compute X, then it's a simple "Add 1 to things $\leq$ X" operation.

Note that computing X is just count distinct numbers range queries, which is standard and present on CSES Problem: Distinct Values Queries It can be done a segment tree and a sweepline.

Solve Count: 153
First Solve: 00:43 by poocha kya

Problem C
Composite Madness

Author: Shreyan Ray

Claim: All $N \geq 15$ is *good*.

Proof: We do casework on $N \pmod 4$.
$N \pmod 4 = 0$, take $\frac{N}{4}$ 4s.
$N \pmod 4 = 1$, take 9 and then $\frac{N-9}{4}$ 4s.
$N \pmod 4 = 2$, take 6 and then $\frac{N-6}{4}$ 4s.
$N \pmod 4 = 3$, take 6 and 9 and then $\frac{N-15}{4}$ 4s.

For small $N < 15$, we can either write a bruteforce, or manually work out the cases to find that $N = 1, 2, 3, 5, 7, 11$ are the only non *good* numbers.

Solve Count: 216
First Solve: 00:01 by ICPCmaxxing

Problem D
Delete if Equal

Author: Shreyan Ray

Core Idea: Let's call a substring a **block**, if all the characters are equal, and it's length is more than 1. It is a **maximal block** if it cannot be extended in either direction. Divide the string into various parts by partitioning across maximal blocks, and analyze the structure.

For example, in string 101110100, we will divide it like 10|111|01|00, where 111 and 00 are the maximal blocks.

You can note that the parts which are not maximal blocks, will just be alternating strings. Thus, the string looks like $A_1B_1A_2B_2....A_KB_KA_{K+1}$, where A_i represents an alternating string (some A_i may be empty), and B_i represents a block.

Now, some observations: (Here edge A_i refers to A_1 and A_{K+1})

Observation 1: Each A_i (which is not an edge), can be reduced and merged with it's adjacent blocks.
For example, suppose we have 11|010|11, we first choose the left 0 and delete it to get 111|0|11 (the 1 merged with the left block). Then, we delete the last 0 too to get 11111.

Observation 2: Depending on the edge character, an edge A_i can either be merged with it's adjacent B_i or it can be reduced to a length 1 string (with only the first/last character of the string), which cannot be improved on.

Observation 3: Each B_i can be reduced to a string of size 2, and no further. In essence, they form a divider across which operations cannot take place.

Implementing this properly is also the solution. Observation 2 and 3 give proof of optimality. The exact steps are as follows:

1. Delete all non-edge A_i, and either delete edge A_i or convert it to it's edge character. (You don't have to actually implement merging because we will be reducing each block to size 2 anyways)

2. Some blocks will get merged in the above process. This happens when both B_i and B_{i+1} are 0 or both 1 blocks.

3. Reduce each B_i size to 2, and output the final length

Solve Count: 142
First Solve: 00:42 by CodeDawgs

Problem E
Expected Rain

Author: Himanshu Singh

First of all, let's find $f(H)$.
Let's do contribution counting. The amount of rain collected over H_i will be $\min(pre_i, suf_i) - H_i$ where pre_i denotes the maximum height of a building $\leq i$, and similarly suf_i. Adding this over all i gives us $f(H)$.

Let's observe one more thing (this is not essential to solve the problem but does help), either $pre_i = \max(H_i)$ or $suf_i = \max(H_i)$. Thus, the formula can be rewritten as $pre_i + suf_i - \max(H_i) - H_i$.

Calculating the expected value of the last formula is easier (but the other is still not impossible). We essentially want $E[\sum pre_i]$, because $E[\sum suf_i] = E[\sum pre_i]$, and $\max(H_i)$ and $\sum H_i$ are fixed.

Let's do yet another contribution counting. With linearity of expectation, we can reduce $E[\sum pre_i]$ to essentially calculating the expected number of times that H_x is a prefix maxima (multiplied by H_x), i.e.

$$\sum_{x=1}^{N} H_x \cdot E[\#i : pre_i = H_x]$$

.
(Actually, this formula is wrong when repeated H_x occurs, you can instead consider the pairs (H_i, i) and calculate pre_i over these).

Now, we will simplify the expression.

$$E[\#i : pre_i = H_x] = P[\exists i : pre_i = H_x] \cdot E[\#i : pre_i = H_x; \exists i : pre_i = H_x]$$

We basically conditioned the expression to first calculate the probability that H_x is at least one prefix maxima, and then calculate the conditional expected value. Clearly this is equivalent

$P[\exists i : pre_i = H_x]$ is simply $\frac{1}{N-x+1}$ assuming H is sorted. This is because, it is essentially the probability that out of $H_x, H_{x+1}, ..., H_N$, H_x is the first to appear. Each number has an equivalent chance of appearing first.

Now, we assume H_x is the first to appear and we calculate the expected value of number of times it appears in the pre_i expression. Let us have a partial sequence only consisting of the numbers $H_x, H_{x+1}, ..., H_N$ (in their order), so suppose order is something like $H_x, H_y, H_z, ...$ where y, z are arbitrary but $> x$.

H_x will remain maxima till we hit H_y, so equivalently, we want the expected number of things (H_i such that $i < x$) that will be present in the gap between H_x and H_y. There are a total of $(N - x + 2)$ equiprobable gaps, where only 1 gap is good for us. We will be filling in $x - 1$ things. Thus, the expected number of such things present in the good gap is $\frac{x-1}{N-x+2}$.

Thus, $E[\#i : pre_i = H_x; \exists i : pre_i = H_x] = 1 + \frac{x-1}{N-x+2}$, the $+1$ because it appears at least once irregardless of the things in the gap. This solves the problem in $O(Nlog(MOD))$ to find inverses.

Solve Count: 41
First Solve: 1:20 by 404_solution_not_found

Problem F
Flip on Cycle

Author: Vikram Singla

First, let us try to find some necessary conditions.

The grid graph is bipartite, and hence all cycles have even length. This means that $\sum(A) \equiv \sum(B) \pmod 2$ is a necessary condition, since each operation flips even number of cells.

Infact this is sufficient admit a few edge cases. Let $C = A \oplus B$. For $n = m = 2$, all $C_{i,j}$ must be either 1 or 0. For $n = 2$, $C_{0,i} = C_{1,i}$ is necessary because the cycles we can choose is highly restricted. Every cycle we choose must include either both $(0, i)$ and $(1, i)$, or neither. Equivalent condition for $m = 2$.

We can ensure that all the cells with $1 \leq i < N, 1 \leq j < M$ have $C_{i,j} = 0$ by just doing an operation with $(i, j), (i + 1, j), (i + 1, j + 1), (i, j + 1)$ if $C_{i,j} = 1$. This shifts all the 1s to the edges of the grid.

Now, we can flip 2 adjacent cells in the edges of the grid by first flipping a 6-cycle, and then a 4-cycle, where the 2 cells not flipped in the 4-cycle are exactly the original adjacent cells.

Solve Count: 14
First Solve: 2:29 by 404_solution_not_found

Problem G
Infinite Operations

Author: Himanshu Singh

Observation 1: Suppose we have some A_i and A_j such that $A_i \pmod 2 \neq A_j \pmod 2$. Then, we can repeat operations on (i, j) till infinity.

Claim 1: For $N \geq 3$, it is always possible to find/create such a pair unless $A_1 = A_2 = ... = A_N$ which is a trivial `No`
Proof 1: If it's not possible to find such a pair, it means all elements have same parity, suppose all even. Choose any 2 non-equal elements, both will become odd now. Choose any other even number remaining (since $n \geq 3$, it exists), and any of the 2 odd numbers as the required pair.

For $n = 2$, we can see that the answer if `Yes` if and only if $A_1 \pmod 2 \neq A_2 \pmod 2$.

Solve Count: 216
First Solve: 00:06 by Seg3

Problem H
Keep it Simple (and Bipartite) Stupid!

Author: Shreyan Ray

Clearly, a player is unable to make a move when the game state is a complete bipartite graph only. At that point, there are $R \cdot (N - R)$ edges in the graph, where R denotes the size of one bipartite component. If we knew the value of R, then the problem was solved because by parity we could tell who makes the last move and who loses.

If N is odd, then $R \cdot (N - R)$ is always even regardless of the value of R. Hence, we can just check whether M is odd, or even; odd implies Alice win, otherwise Bob.

If N is even, then $R \cdot (N - R)$ has the same parity as R. If the graph was entirely one component, we would know the value of R. On the other hand, if the graph has multiple components, then it is trickier as the way we combine multiple components matter.

Some preliminary observations. Let's call components OO, OE, or EE depending on the parity of number of nodes in their 2 bipartite parts. Merging is essentially deciding what part to add to what. For example,
OO + OE = EO = OE
OE + OE = EE
OE + EO = OO

Observation 1: The way we combine OO or EE components do not matter, but the way we combine Oe does.

Observation 2: The number of OE components remain conserved under merging unless you combine 2 OE components (where they will reduce by 2).

Observation 3: We can *nearly* always merge OE and OE to get either OO or EE. The exception is 2 singleton components.

Observation 4: Suppose that there are no OE components. Then, we can see that the parity of R is fixed, and thus the game is forced win for one player or the other depending on the parity.

Observation 5: Suppose there are exactly 2 OE components, at least one of which is not a singleton. Then, note that Alice on her first move can combine them in 2 ways to get different parity states. Exactly one of the states will be winning, the other will be losing. Alice can make sure Bob receives the losing state, hence, she wins.

By the above observation, we have that only 2 OE components (not both singleton) is always winning.

Therefore, a player would not want to give their opponent such a state. Let's ignore singletons for now, and analyze when we are **forced** to give our opponent a state with 2 OE components. Let's call such a forced state as **Saturated**.

Any saturated state will have exactly 4 OE components, and no possible waiting moves. This means that each component is a complete bipartite graph (as otherwise we can add edge to this), and there are exactly 4 components total (no extra components).

The crucial observation here is that such a state **must have an even number of edges**. Hence, we can argue by parity, that the player who receieves such a state is fixed, i.e. if M is even initially, Alice receives such a state eventually, otherwise Bob.

Let's now generalize to singleton components. Don't count singletons as OE for now, and let s denote the singleton count.

By induction, we will prove the following 2 statements simulatenously:

- State 1 : If there are no OE components (note this means s is even), the game is decided by the parity of $OO + \frac{s}{2} - M$.
- State 2 : If there is 1 or 2 OE components, the game is always winning for first player.

From State 1, there are 4 possible types of moves,

- Add an edge to some component, not uniting any components
- Unite 2 singletons
- Unite 2 non-singletons
- Unite 1 singleton and 1 non-singleton

Note that the first 3 types of moves changes the parity of $OO + \frac{s}{2} - M$ by exactly 1. Thus, this is according to our analysis.

On the other hand, the 4^{th} move leads you to State 2 which is winning for your opponent. Thus, you will not want to do such a move.

For state 2, we will combine the 2 OE components (or 1 OE and 1 singleton). We can decide the parity of $OO + \frac{s}{2} - M$ afterwards and ensure that this state is losing for Bob, and winning for us.

When there are more than 3 OE components, we again look at **Saturated States**. We can again observe that Saturated states will have an even number of edges, so the winner is fixed.

Solve Count: 4
First Solve: 2:24 by HackStreet Boys

Problem I
Majority Partition

Author: Matthew Roh

Fact 1: The majority element of an awesome sequence is always unique. Let's associate each sequence with it's majority element.

Observation 1: If X is a majority element of A and X is a majority element of B; then X is a majority element of $\texttt{concat}(A, B)$. Thus, each sequence in an optimal partition has unique majority element.
Furthermore, using this, it can be shown that if X is a majority element of some subsequence, all the occurrences of X lie in that sequence.

Now, we can realize a greedy strategy, which is simply to take the majority elements in decreasing order of $freq_x$. We can formally prove this with exchange argument.

When we choose x to be a majority element of a subsequence, we see that it adds $freq_x$ elements on it's own, and it can add another $freq_x - 1$ elements without changing the majority element. We simply look for the smallest prefix such that $\sum 2 \cdot freq_x - 1$ exceeds N. Time complexity is $O(Nlog(N))$ or $O(N)$ with counting sort.

Solve Count: 211
First Solve: 00:06 by poocha kya

Problem J
Make Equal Under Mod

Author: Shreyan Ray

Observation 1: The set of numbers that X can be changed to by using arbitrary $\pmod Y$ operations is $[0, \frac{X}{2})$ and X itself.
Proof: Numbers larger than $\frac{X}{2}$ can be proven to be impossible by casework on $Y \geq \frac{X}{2}$ or not. We can obtain each number Z in that range by simply choosing $Y = X - Z$.

Now, we are set to find $f(A)$.
$min(A)$ is clearly an upper bound. It is achievable if and only if for all $A_i \neq min(A)$, $A_i > 2 \cdot min(A)$. This comes from using Observation 1.

Otherwise, we can observe that the answer is $\lfloor \frac{min(A)-1}{2} \rfloor$. This is clearly the next best bound on the answer, and it is achievable.

Thus, we have a characterization of $f(A)$ and we need to find the sum over all subarrays.

Let's fix the minimum element A_i, and look in the interval where it is min. If an interval contains a number in the range $[A_i + 1, 2 \cdot A_i]$, then the answer is $\lfloor \frac{min(A)-1}{2} \rfloor$ as discussed above. Otherwise, the answer is $min(A)$.

We will count number of subarrays which contain a number in the range $[A_i + 1, 2 \cdot A_i]$ (and still have A_i as the minimum) and then subtract that from the total number of subarrays with A_i as minimum.

This can be done if we calculate 4 integers L_1, R_1, L_2, R_2, where L_1 is the first integer smaller than A_i to the left, L_2 is the first integer between $[A_i + 1, 2 \cdot A_i]$ to the left of i (and similarly for R).

Calculating L_1 and R_1 is well known task with stack. For calculating L_2 and R_2, we can sweepline in increasing order of A_i, and maintain a sliding interval of numbers in the range $[A_i + 1, 2 \cdot A_i]$, and keep a set of active numbers. Then, we just query the set for the first integer before i and first integer after i.

The whole task can be solved in $O(NlogN)$.

Solve Count: 41
First Solve: 1:28 by SubtasksWhere

Problem K
Max Mod

Author: Chongtian Ma

Note that, at any point, the array A will satisfy $A_i = x + iy$ for some x, y. So, the task at hand is to compute:

$$\max_{i=L}^{R} ((x + iy) \pmod M)$$

WLOG, we can assume $x = 0$, else, we can replace $[L, R]$ by $[L + xy^{-1}, R + xy^{-1}]$, where y^{-1} denotes the modular inverse of y wrt M. Hence, we need to maximize over $i \in [L, R]$:

$$(iy \pmod M) = iy - M \left\lfloor \frac{iy}{M} \right\rfloor$$

Note that, for a fixed $k = \lfloor \frac{iy}{M} \rfloor$, the above quantity is maximized for the largest such i, which is $f_k = \lceil \frac{(k+1)M}{y} \rceil - 1$, and the corresponding value would be:

$$\begin{aligned} f_k y - Mk &= y \left(\left\lceil \frac{(k+1)M}{y} \right\rceil - 1 \right) - Mk \\ &= (k+1)M + \big(-(k+1)M \pmod y \big) - y - Mk \\ &= M - y + \big(-(k+1)M \pmod y \big). \end{aligned}$$

Let $P = \lfloor \frac{Ly}{M} \rfloor$ and $Q = \lfloor \frac{Ry}{M} \rfloor$. Note that for $k = Q$, the best possible i is R. So, first account for that. Then, clearly for all $k \in [P, Q-1]$, $f_k \in [L, R]$. So, we need to maximize $((-M)k)$ mod y over all $k \in [P+1, Q]$.

Hence, we can recurse into smaller subproblems. This time complexity is good if $y \leq \frac{M}{2}$. Otherwise, note that we need to maximize $(R - i)y \mod M$ over $i \in [0, R - L]$. Or equivalently $(Ry + (M - y)i) \pmod M$. Hence, we can replace y by $M - y$. Thus, there will be $O(\log M)$ iterations.

Thus, the problem is solved in $O(Q \cdot log(M))$ or $O(Q \cdot log^2(M))$ depending on implementation.
Solve Count: 0
First Solve: -

Problem L
Ticket Revenue Maximization

Author: Jatin Yadav

Each round can be viewed as balanced bracketed sequence, where a (means they lost the round, and) means they won the round. Let us explain why this characterization is both necessary and sufficient.

First of all, the number of winners must be equal to losers of course. Suppose some suffix has positive balance [looking at (as +1,) as -1]. Then, there are more losers than winners here. But that is impossible as they cannot be paired up with that many stronger people.

Thus, every suffix has negative balance (or 0), and equivalently, every prefix has positive balance. Hence, it is necessary for it to be a balanced bracketed sequence, but is it sufficient?

We know in a balanced bracketed sequence, we can pair up the brackets into pairs of () such that the (always appears before the). This also gives us a valid pairing in terms of matches.

Now, let's extend this idea to multiple rounds. Let's call a person a x-loser if they lose in round x, and a person a x-winner if they win in round x.

The necessary and sufficient condition for the tournament to be valid is in every suffix, there are at least as many x-winners as x-losers.

We can do *dp* with our states being the number of x-losers for every x from 1 to 7, from back to front. Then, for the next person, we can decide which round we want to make him lose at, while ensuring the condition of every suffix having at least as many x-winners as x-losers is still true. The strongest person will always be the winner of all rounds.

Naively, this has $65 \cdot 33 \cdot 17 \cdot 9 \cdot 5 \cdot 3 \cdot 2$ states, which is around 10^7, however, you can note that a large amount of these are useless. This is because the suffix condition of more x-winners $\geq$ x-losers does not allow all states. The actual number of states is around $2 \cdot 10^5$.

Recursion with proper memoization, by unrolling the information into an array, can pass in ≤ 0.5s, while just putting the vectors into a map for memoization passes in around 2.5s.

Solve Count: 0
First Solve: -