A

A. And to Zero

When the AND of all numbers is positive, then answer is -1. In other cases we are able to obtain some 0. In some solution let's look at a zero which ends the game. It is result of the AND of some subset of numbers from the array. It appears that number of sequences of moves for every (smallest) subset is the same and is equal to $k! \cdot (k-1)!$ where k is the size of this subset. It's because when subset is fixed for every move we should choose two of it's elements, do a move on them and forget about the one that haven't changed, so it's a simple combinatorics.

A. And to Zero

Now we want to count such subsets. We need an observation that minimal size of subset won't be greater than 20, because in every move the result should "lose" some bit.

Let's check if there is some subset for some fixed k. Some simple DP should come to mind, where for each value of the AND of a subset we would calculate smallest size of subset with this AND and number of such subsets. Unfortunately this DP works in $O(m^2)$ time, where $m = 2^{20}$, so it's too slow.

Let's modify it a little bit and change state to $DP_{k,AND}$, which will denote number of subsets of size k with some fixed AND (may be equal to 0). This can be speeded up with Fast Subset Convolution, which does exactly what we want.

A. And to Zero

This would work in $O(m \cdot log^2(m))$ which is still too slow. A trick is that we don't have to reverse convolution $O(log(m))$ times, because we are not interested in all values, only in the one with the AND equal to 0, and we can calculate this value in $O(m)$ instead of $O(m \cdot log(m))$. So overall complexity is $O(m \cdot log(m))$.

A. And to Zero

This isn't the end, there is a tricky case. We are calculating our answer modulo $10^9 + 7$, but there is possibility, that number of subsets is divisible by this modulo and isn't equal to zero. What we have to do is to pick another modulo (any one, if you realized this problem, you should get an AC), calculate answer modulo this second number and use it to check whether the number of subsets is still equal to zero.

B

B. Bitwise-Xor to Zeroes

It turns out that if we want to turn some interval into zeroes, then optimal strategy goes as follows:

If there are already some zeroes in the array, then don't touch them.

As long as there is a pair of equal positive values, then erase them both in one move.

If there is no such pair, then just erase each value separately.

So if some value x occurs a_x times in our interval, then we should add $\lceil \frac{a_x}{2} \rceil$ to our result. All answers can be easily gathered with MO algorithm in $O(n^{1.5})$ time.

Also squeezed $O(n * q/32)$ solution should pass.

C

C. Cutting Tree

There are an $O(n \cdot log^2(n))$ and an $O(n^{1.5})$ solutions, but here will be described an $O(n^{1.5} \cdot log(n))$ one, which with some good ideas can be much faster than previous ones.

C. Cutting Tree

Firstly let's invent a solution for one k. It can be proven that this is optimal to "cut" any subtree of size not less than k (we can assume that component can be greater than k) such that it has no child which has a subtree with size also not less than k. This can be easily done in linear time and we'll use it as a blackbox.

C. Cutting Tree

For some fixed k anwer cannot be greater than $\frac{n}{k}$. Obviously there are $\sqrt{n}$ ks lower than $\sqrt{n}$ and for every k greater than $\sqrt{n}$ the answer will be lower than $\sqrt{n}$, so the answer can only have $O(\sqrt{n})$ different values.

So when we have some k we want to check for how long the anwer wouldn't change if we'd start to increasing k. We can use binary search to find this moment. With previous knowledge we know that there will be at most $O(\sqrt{n})$ binary searches, so solution will use blackbox $O(\sqrt{n} \cdot log(n))$ times.

C. Cutting Tree

This may not fit into time limit.

First thing to optimize it is to write blackbox without recursion, so it will be much faster.

Second thing is that binary searches are independent (they are not using knowledge from previous blackbox calls). Best way to fix it is to make a recursive function $solve(a, b, low, high)$ which has to calculate answers all ks from range $[a, b]$ with knowledge that all the answers will be in range $[low, high]$. If $low = high$ then function will know that all anwers in range $[a, b]$ are equal, otherwise it will ask blackbox about k in the middle of this interval and call itself on two halves with new bounds.

C. Cutting Tree

Hint for an $O(n \cdot log^2(n))$ solution: if an answer for each k doesn't exceed $\frac{n}{k}$, then sum of all answers will be at most $O(n \cdot log(n))$, so maybe there is some way to get componens from answers one by one, each of them in $O(log(n))$?

D

D. Dance Classes

It can be observed that if both arrays are random then every matching also has random value (incompatibility). So our task is to just look at as many matchings as we can.

If we'll check 14! matchings then chance that there will be no matching with value equal to 0 are small enough, so this will be our aim.

D. Dance Classes

If n is not greater than 14 we can check every possible matching with some meet-in-the-middle. Let's iterate over subset of girls which will dance with first half of group of boys. Then iterate over every permutation of these subset of girls and then over every permutation of rest of girls. With some sorts and binary searches it's possible to find an optimal answer.

If n is greater than 14 then just do this trick for any subset of 14 girls, for sure you'll find some answer with value equal to 0.

Complexity is $O(\frac{m!}{(\frac{m}{2})!})$ where $m = min(n, 14)$.

E

E. Exact Covering

Let's firstly use $\sum weights$ paths to cover the edges. Now we'll just merge them in vertices. Each merge obviously decreases number of paths by 1.

It can be observed that merges in different vertices are independent from each other.

Moreover, number of merges which we can do in some vertex depends only on sum of weights of edges incident to this vertex and on maximum one of this weights.

E. Exact Covering

So solution should remember sum of weights of edges incident to every vertex and also multiset of these weights to be able to answer queries about maximum one in $O(log(n))$ time. Overall complexity is $O(n \cdot log(n))$.

F

F. Foreheads Game

Let's generate every possible set of cards (along with their assignment to players). Now let's calculate for each turn which sets will fall out of game in this turn. We know which turn it is, so we know which player will now say something.

For every set of cards we know which card isn't known for currect player, so this player has some options for this card. Some of this options aren't possible, because for them we (and this player) already know that somebody would already end the game.

If in all possible options this player has this same rank, then he will end the game, so for this set we should remember that it already isn't possible to have this set still in game.

G

G. Grid with Mirrors

Let's build a graph where vertices are rows and columns of the matrix. If some row/column is splitted into several rows/columns by blocked cells, then we'll also split it into several vertices.

Now we want to run a bfs on these vertices, starting it from row which contains cell $(1, 1)$. Two vertices are connected with a bidirectional edge if one of them is a row, second is a column and they have a common cell. Then going through this edge stands for putting a mirror in this cell so that light beam from row/column will change direction to column/row.

G. Grid with Mirrors

The problem is that we cannot just add edges, because there are too many cells in the grid.

One way to get this graph is to use persistent tree and sweep over some dimension. Our tree will be used to speed up bfs, because every vertex of tree will contain edges directed to its children. Later let's do the same with second dimension.

The graph will have $O((n + m) \cdot log(n))$ vertices, will become directed and it will be possible to run bfs on it.

G. Grid with Mirrors

When we know for every row and every column its distance from the first row, we can calculate the answer using fact that for each cell the absolute difference between the number of mirrors needed to light up its row and the number of mirrors needed to its column is exactly 1.

H

H. Heavier Coins

Let's build a graph where coins are vertices and weightings are directed edges. It turns out that if we'd for every vertex calculate length of longest path which starts in this vertex and sum up these values it'll be an answer for the problem.

H. Heavier Coins

Proof, part 1:

Let's sort our graph topologically (it's a DAG because there are no contradictions). Then let's choose a vertex v which is earliest in this topological sort. We can start with one vertex which is the end of longest path which starts in v and successively exchange it with vertices on this longest path until we'll come to vertex v. For sure weight will be increasing all the time. v will now stay in our subset forever and we'll solve rest of the graph recursively. With this construction we can achieve such result.

H. Heavier Coins

Proof, part 2:

It's possible that weigth of every coin is equal to length of longest path which starts in this coin. Then weigth of every subset is an integer from range $[1, \sum longestpaths]$, so result can't be better.

H. Heavier Coins

Answer won't be greater than $\frac{n \cdot (n+1)}{2}$, so modulo is not needed in this problem. It doesn't change the fact, that you cannot forget about it. ;)

I

I. Isomorphic Matrices

Let's assume that n is not greater than m, if it isn't, just swap them. Now n is lower than 45.

I. Isomorphic Matrices

We'll use Burnside's lemma. For each permutation of rows and each permutation of columns we should calculate number of coloring that won't change after applying these permutations.

Each permutation of rows and columns results in some permutation of cells. It turns out that if some cycle in permutation of rows has length x, and some cycle in permutation of columns has length

y, then these two cycle result in $gcd(x, y)$ cycles in premutation of cells.

We are only interested in number of cycles in permutation of cells, because the value that we want to sum up for a pair of permutations (rows and columns) is equal to $k^{number of cycles in permutation of cells}$.

I. Isomorphic Matrices

From this formula we know that we are only interested in multiset of lengths of cycles in permutations of rows and columns.

We can iterate over this multiset in permutation of rows because n is lower than 45. There will be about 75000 of such multisets.

We cannot iterate over multisets of lengths in partition of m, cause m can be much bigger. Instead of this we'll calculate DP, where DP_v is the sum of values that we want to sum up if m would be equal to v instead.

To calculate this DP we should iterate over all possible length of cycle that contains number 1 and if this length is equal to i then we should to DP_v add DP_{v-i} multiplied by some factorials and also by $k^{sum of gcds with lengths in partition of n}$.

I. Isomorphic Matrices

Last part is to calculate for each multiset of lengths for how many permutation does it correspond. It can be done with some simple combinatorics.

Overall complexity is $O(m^2 \cdot (number of partitions of n))$.

J

J. Just Xor Partitions

Let $prefixxor_i$ denote the XOR of first i elements from the array.

If it's possible to divide the array according to given rules, then all $prefixxor$s of places where intervals end, must be achievable by taking the XOR of some subset of numbers from the set from query. There are at most 32 such subsets, so we'll run Dijkstra on them and for each of them we'll calculate length of shortest (but non-empty) prefix of the array which has $prefixxor$ equal to the XOR of this subset and is splittable into good intervals.

J. Just Xor Partitions

We need some data structure, which will be able to change elements of the array and for some given x and y will be able to find smallest (but not smaller than x) i, such that $prefixxor_i$ is equal to y.

It's possible to do sqrt-decomposition here. We'll divide our array into $\sqrt{n}$ smaller arrays and for each of them we'll keep some value (offset) and remember that all elements from this small array

should be XORed with this special value. During query of first type we'll completely update only one small array.

To avoid having extra $log(n)$ factor we can keep $\sqrt{n}$ bitsets of size 2^{20} and store in them the information which $prefixxor$s are in this small array.

J. Just Xor Partitions

Complexity for changing element is $O(\sqrt{n})$ and for query about splitting array is $O(\sqrt{n} \cdot 2^k)$.

You may think that there should be some k factor, but we can observe that we should only consider first time when we try to get into XOR of some subset of values from input.

K

K. KMR

Inventing a solution for version of task with only two teammates is rather easy, DP should just remember in which states are them. Unfortunately, if we'd try to improve this solution for bigger number of teammates it will become $O(n^2 \cdot m)$, which is too slow.

The hint is that this solution should be left and some new one should be invented. Maybe there is a way to solve it even for bigger number of teammates?

K. KMR

It turns out that the answer is yes. Let's firstly calculate number of paths between every pair of vertices and denote this number by $paths_{v,u}$ (for paths from vertex v to vertex u). This can be easily done in $O(n \cdot m)$ with simple DP.

Now, let's calculate DP_v, where DP_v is the number of ways in which we can choose 3 paths, such that they start in vertex 1, end in vertex v and there is no edge which lies on every path. Of course DP_n will be the answer to the problem. How to calculate it? Number of all triples of paths is simply $paths_{1,v}$, but we have to subtract bad triples. Let's iterate over which edge was first bad edge in some bad triple. Let's say that it was edge from vertex u to vertex w, then we have to subtract $DP_u \cdot paths^3_{w,v}$. Overall complexity is $O(n \cdot m)$.

L

L. Lazy Kids

Two players are taking random balls from a bag with red and blue balls, the first to take a red ball loses. Determine the probability of the first player losing.

L. Lazy Kids

There are many possible approaches to this problem:

- We can compute the answer $P_{r,b}$ with DP, since $P_{r,b} = \frac{r}{r+b} + \frac{b}{r+b}(1 - P_{r,b-1})$.
- We can go with a purely combinatorial solution. Suppose that k blue balls were taken before the first player took a red ball. The probability of that is $\frac{b!/(b-k)!}{(r+b)!/(r+b-k)!} \cdot \frac{r}{r+b-k}$. We should sum these values over all even k.

Note that the answer may be long, hence you should implement long arithmetics, or use Python or Java.