Problem A. Saving the cinema

You should do what the problem says with an if statement. Be careful to add the final dot.

Problem B. Operation

First we need to realize that the substraction and division of positive integers just make the result lower, whereas the multiplication and addition do not.

So since there are just two possible options to optimize the solution we can just compare both of them and output the bigger one.

Problem C. Maximum profit

To solve this problem we need to sort the array in non-decreasing order and then calculate the sum of the last k elements.

Complexity of the solution: $\mathcal{O}(n \cdot log_n)$.

Problem D. jbum

To find m, that is the mininum amount of minutes Javier needs to wait until he can perform with the desired weight, we just need to calculate $\lceil log_2(n) \rceil$, because in the i-th minute we can add $2^{(i-1)}$ discs if we always put in the duplicator all the discs we have.

Once we know m, we need to find the lexicographically smallest sequence of x, where x_i is the number of discs that Javier puts into the *duplicator* in the i-th time. We will call a_i the amount of discs we have in the i-th minute. So we have to find an array a that satisfies $a_i \geq 2 \cdot a_{i+1}$, but as we need to find the lexicographically smallest between of all possible x, a also needs to satisfie that $a_i = \lceil \frac{a_{i+1}}{2} \rceil$. This happens beacuse if $a_i > \lceil \frac{a_{i+1}}{2} \rceil$ it means that we could decrease a_i, until it becomes equal to $\lceil \frac{a_{i+1}}{2} \rceil$. And as we know that $a_r = n$, we can calculate how many discs Javier needs to put in the *duplicator* in each minute, first calculating the $(m-1)$-th one, and ending knowing how many discs we have to put in the *duplicator* in the first minute.

Problem E. Looking for palindromes

The answer is $10^{\lceil \frac{n-2}{2} \rceil} \cdot 90 \cdot \lceil \frac{n-1}{2} \rceil$. This formula is basically the number of palindromes of length $n-2$, and multiplying by the positions and combinations of the newly added digits.

Problem F. Harry Potter in CMS

This problem was solved using the *set* data structure, or another Balanced Binary Search Tree.

If we let s be the set which stores the first query that gets each subtask correct, the answer the size of s. The idea was to store, for each subtask, the queries in which Harry got that subtask as correct, and add the first of those queries to s. Then, for the removing query, we update the set of the subtasks that were correct in that query and update set s accordingly. This part was mostly implementation, we recommend looking at the code if you still have doubts.

Final complexity: $O(q \log q + k \log k)$.

Problem G. XOR + Constructive = Love

Firstly, instead of using the array b mentioned in the statament, we will use another array cnt which will count the amount of elements that have each bit, similar to the described array a in the statement, which is easier to work with. Because of the first condition, for all i, cnt_i should be at least a_i, so we initially set $cnt_i = a_i$ for $i < 30$, and $cnt_i = 0$ for $i \geq 30$. Then, it is also easy to check the XOR condition for each bit. If bit i isn't satisfying the XOR condition, we add 1 to cnt_i.

Once we have done this, we should only worry about meeting the sum condition, knowing that we have to add bits in pairs, i.e. the parity of each cnt_i should remain constant.

Now, one of the main observations is that, if there exists an answer, the answer will be at most $\max\limits_{0 \leq i < 60} cnt_i + 2$. This makes it easier because you can iterate over the possible sizes and check if there exists a solution of that size.

To do this, let the current size be m and let $left$ be $s - \sum\limits_{0 \leq i < 60} cnt_i \cdot 2^i$. If $left$ is negative, it is impossible to fulfil the condition. Otherwise, we just iterate over the bits from most significant to less significant, and add as many as we can. Because we don't want to exceed $left$ and we can only add at most $m - cnt_i$ to each bit i, we could add $\min(\lfloor \frac{left}{2^i} \rfloor, m - cnt_i)$ to bit i. Additionally, we should check once again if the XOR condition is met (it would stop being fulfilled if we added an odd number of ocurrences for a bit), and if it isn't, substract 1 to the amount we are adding. We update $left$ and repeat the process for the next bit. If at the end of the process $left = 0$, then it is possible to build an array b of size m.

If we didn't find any possible size, we should print -1.

Final complexity: $O(\log s)$.

Problem H. Menorca's ants

Let's imagine that we have a subset of x ants where the maximum number of ants of the same type is y and we want to know the minimum number of throws needed to throw all the ants.

The m condition depends on x and the k condition depends on y. It can be proven that the answer will be $max((x + m - 1)/m, (y + k - 1)/k)$.

Proof:

$(x + m - 1)/m =$ minimum number of throws needed without the k condition, and it doesn't depend of the order of the throwed ants.

$(y + k - 1)/k =$ minimum number of throws needed without the m condition, and it depends of the order of the throwed ants.

We can always use the optimal order to minimize the maximum number of ants of the same type used and we will have enough moves.

The last thing we need is to find a subset of p ants whith the maximum number of ants of the same type minimized. To do this we can just binary search on the answer (z) and checking if the sum of $min(z, a_i)$ for every i is $\geq p$.

There are other ways to do this, for example with sorting.

Complexity of the solution: $\mathcal{O}(n \cdot log_A)$ where A is the maximum a_i among every i.

Problem I. Fake bills

The main observation for this problem is that the state of the room is cyclic every 4 turns, because the cameras return to their initial position.

With that observation, you can store if each cell will be covered by a camera or not for each of the 4 states, so you can keep states of the form (t, r, c), where t is the time that has passed (denoting the state of the cell), r is the row and c is the column. To store this information, it is important to note that you can't just iterate over the cameras and then over the cells that each camera covers, because this has a complexity of $O(m \cdot n)$, which is too slow. Instead, you should keep the minimum and maximum coordinate of any camera pointing in each direction for each row or column, and then determine if a cell is covered in a state using this information in $O(n^2)$.

To do this, if a camera points up, you update the maximum of that column with the row of the camera. If it points down, you do similarly with the minimum of that column. If it points right, you update the minimum of the row with the column of the camera, similarly with the maximum if it points left. Then, a cell at (r, c) is covered if and only if:

- $\max_r \geq c$ or,
- $\min_r \leq c$ or,
- $\max_c \geq r$ or,
- $\min_c \leq r$

Then, you do a BFS or DFS of the room, starting at $(0, 0, 0)$ and going to adjacent cells with $(t + 1) \bmod 4$ as state which aren't covered by any cameras.

If you reach cell (n, n) at any point, a robber can reach the vault. Otherwise, it will be impossible for him.

Final complexity: $O(n^2)$.

Problem J. Force Perturbation

This problem is solved using knapsack DP. Calculate, for each number of elements, the nearest sum to x that it is possible to make without using an element twice. Let that sum for a number of elements i be denoted by dp_i.

Then, the answer is $\min_{i=1}^{n} \lceil \frac{x - dp_i}{i} \rceil$.

Final complexity: $O(n^2 \cdot x)$.

Problem K. Óscar and his battle

Firstly, we sort the characters by their attack and the monsters by their defense. Then, we iterate over the characters. Everytime we get to a new character i, we process all the monsters j such that $d_j \leq a_i$. This is done to two pointers to keep a complexity of $O(n \log n + m \log m)$, due to the initial sorting. When we process a monster, what we do is add e_j to the index c_j of a Segment Tree or Fenwick Tree.

Once we have processed all the monsters, to know how many coins we could win with a character i, we just have to query the sum of elements less or equal to b_i in our data structure. This is because, due to the initial sorting, we know that all the monsters in the data structure have a defense lower or equal to the attack of the current character, and then we only sum the coins given by monsters with an attack lower or equal to the defense of the current character, so we know that the character will be able to defeat all of them.

We keep the maximum of the answers for each character as the final answer to the problem.

Note that, due to the magnitude of the stats of the players and monsters, one should coordinate compress the values or use a sparse data structure. Briefly put, coordinate compressing x values means assigning a unique id between 1 and x to each different value, preserving their relative order.

Final complexity: $O(n \log n + m \log m)$.

Problem L. Random intervals

First of all we need to check if the given intervals intersect, this can be easily done by sorting the intervals.

To solve this problem we need to calculate a $dp_{[i][j]}$ = number of ways to put i intervals on the first j spaces between existing intervals, $O(n^2)$ states with $O(n)$ transitions.

If we have already calculated $dp_{[i][j]}$, we can make transitions of the following form: $dp_{[i+k][j+1]}$ += $dp_{[i][j]} * W$ for $0 \leq k \leq n - i$, were W is the number of ways to put k intervals on the space $j + 1$.

To calculate the dp we need to know that the number of ways to put a intervals in a space of length b is $\binom{b+2\cdot a-1}{2\cdot a}$. You can get more info by searching for balls and boxes combinatorics on internet.

Then we need to erase the duplicates, this happens when we have intervals of length one because we can put the same interval either at the left or at the right. To do this we just calculate for each space $dp2_{[i][j]}$ = number of ways to put i intervals in that space with j intervals of length one at the left.

We also need to precalculate factorials and inverse factorials.

Complexity of the solution: $\mathcal{O}(n^3 + m)$.

Problem M. The battle of Helm's Deep

The first thing to notice is that the towers are independent between them, the state of a tower doesn't affect another tower, just the final damage. So, this leads us to think that we can compute, for each tower, how much damage the inner walls would take if we assign some number of soldiers to it.

Additionally, an important observation is that the damage the inners walls take from a tower depends only on when the tower falls, so we can compute how many soldiers we need to assign to a tower i so that it falls in wave j. As there are a total of q waves over all towers, this means that we only need to compute this information $O(q)$ times. So, for each tower, we will iterate over the waves that attack this tower and maintain the minimum number of soldiers needed for the tower to fall in that wave (special consideration for the case of when a tower doesn't fall at all). There are multiple ways to do this, some easier and some harder, the author does it in $O(m + q \log q)$. However, an easier solution in $O(m \cdot q)$ which also passes is for each number of soldiers, iterate over all the attacks on that tower in order, and calculate when it would fall. The code uses this approach, as it is easier to understand.

With this information, we can easily calculate $dmg_{i,k}$, denoting the damage the inner walls would take from the i-th tower if k soldiers are in it. This will be helpful later.

Once you have calculated the minimum number of soldiers needed for the tower to fall in some wave j, you can convert that to some "weights" of the form (soldiers, damage), where damage is $q - j$, and do knapsack DP with them in $O(m \cdot q)$. The knapsack DP would have states $dp_{i,k}$, which denotes the minimum damage the inner walls can take if we consider the **last** i towers and we place k soldiers in total.

We can already know the minimum damage the inner walls can take, given by $dp_{n,m}$, but we now need to construct the lexicographically minimum such sequence for which the inner walls take that damage. To do this, we iterate from the first to last tower, maintaining two variables s and d, denoting the soldiers we have already used and the damage we have already taken. Then, for a tower i (1-indexed), we iterate over the number of soldiers we want to place in it, and we place the minimum number of soldiers k such that $d + dmg_{i,k} + dp_{n-i,m-s-k} = dp_{n,m}$.

Note: The code has a slight twist, instead of having $dp_{i,j}$ denoting the minimum damage considering the last i towers if j soldiers are placed, it denotes the minimum damage placing j soldiers in towers from i to n. The former was described in the editorial because it seems easier to understand.

Final complexity: $O(m \cdot (n + q))$.

Problem N. The Omer's orange tree

We will solve this problem by processing the queries offline and doing a sweepline. That is possible because $\sum_{i=a}^{b} f(u,i) = \sum_{i=1}^{b} f(u,i) - \sum_{i=1}^{a-1} f(u,i)$. So, we will iterate over each j such that $1 \leq j \leq n$ and maintain a data structure such that we can calculate $\sum_{i=1}^{j} f(u,i)$ quickly for any u. Additionally, we will store the answer of each query in an array ans, so that ans_k is the answer to the k-th query, which we will update over the sweepline.

The first thing we should do is flatten the tree into a one dimensional array using the Euler Tour Technique, so we can calculate the sum on the subtree of some node in $O(\log n)$ by using a Fenwick Tree or Segment Tree.

Once we have computed the Euler Tour array, we will iterate over all j such that $1 \leq j \leq n$. Firstly, we will need to update our data structure taking into account the new j, so we will iterate over all multiples of j and add 1 in the position in the Euler Tour array of the node with a weight equal to that multiple. Once we have done this, we can calculate $\sum_{i=1}^{j} f(u,i)$ for any u in $O(\log n)$ time by doing a range sum query on our data structure. Now, we will iterate over all the queries of the form (u,a,b) such that $j+1=a$ and for each query k, substract $\sum_{i=1}^{j} f(u,i)$ from ans_k. Similarly, we will iterate over all the queries such that $j=b$ and for each query k, add $\sum_{i=1}^{j} f(u,i)$ to ans_k.

To calculate the complexity of the solution one must take into account that, in total, we will visit $\sum_{i=1}^{n} \frac{n}{i}$ multiples. That sum is known as the n-th harmonic number, which has a value of approximately $n \ln n$. Because we do a point update query on our data structure for each of those multiples, the complexity of this part is $O(n \log^2 n)$.

Note that it can also be solved online using merge sort tree.

Final complexity: $O(n \log^2 n)$.

Problem O. Bea the maximizer

To solve this problem we will convert the arrays in a bipartite graph named G, nodes in the left will be positions on array a and nodes in the left will be positions on array b.

Let's define sz_x as the cardinality of the maximum bipartite matching if there is an edge between two nodes u and v if $a_u + b_v$ is a submask of x.

First we will construct the answer bit per bit, in decreasing order of bits. When checking if we can set a certain bit i having a current solution x, we will check if $sz_{x+2^i} = n$.

After doing this, we will know the maximum possible value, and to find the minimum maximum distance we will do a binary search of the distance and remove the edges of the graph between two nodes u and v if $|u - v|$ is greater than the value being checked, then we will check if the size of the maximum bipartite matching is n.

Both Khun's algorithm and Hopcroft Karp can get AC.

Complexity of the solution: $\mathcal{O}(n^3 \cdot \log A)$ where A is the upper bound of the values in the arrays.

Problem P. Ski resort

The first observation in this problem is that we can compress the graph into a graph of $O(k)$ nodes with weighted edges, the nodes are the nodes from which there is an outgoing ski lift, which we will denote by *special* nodes, and the edges are the maximum number of seconds you can spend going from a to b by using **exactly** 1 ski lift. To calculate the weight of the edges, one can store, for each node, an array $dist$, where $dist_{i,j}$ denotes the distance from node i to the j-th *special* node.

To compute this array, as the graph is acyclic, you can start a BFS from the nodes wich have no outgoing edges. If you currently are in node u, you compute $dist_u$ in the following way: if u is the *special* node number i, then $dist_{u,i} = 0$, if it is not a *special* node, we don't do this step. Then, for each outgoing edge from u to v, for each i, $dist_{u,i} = \max(dist_{u,i}, 1 + dist_{v,i})$.

After computing $dist_u$, for all edges from some v to u, you substract 1 from the outdegree of v, if the outdegree of v becomes 0, you add it to the queue.

This algorithm ensures that the distance is always maximal, as you are considering all the possibilities and taking the maximum out of all of them, due to the order in which it is done.

Similar algorithms could work, for example using the topological order of the graph. The important aspect is that all the children of a node are processed before it.

For the representation of the compressed graph, we will use a matrix A, $A_{i,j}$ denoting the maximum distance from the i-th *special* node to the j-th *special* node, while using one of the outgoing ski lifts from the i-th *special* node.

Once we have calculated $dist_{i,j}$, we can calculate the matrix A by iterating over the ski lifts. Let the current ski lift go from a to b, then for all *special* nodes c reachable from b, $A_{a,c} = \max(A_{a,c}, 1 + dist_{b,c})$.

Now that we have calculated this matrix, it is easy to observe that we can easily get the distance from any special node to any other special node using k lifts by "merging" A with itself $k-1$ times. We use "merging" instead of exponentiating A to k, because the operation is not exactly the same as matrix multiplication, although the algorithm for computing it and the complexity are the same. Instead, if we "merge" two matrices A and B of size $k \times k$, in this problem we need the resulting matrix C to be defined in the following way:

$$C_{i,j} = \max_{1 \le q \le k} A_{i,q} + B_{q,j}$$

Even if is not exactly matrix multiplication, we will denote A merged with itself $k-1$ times as A^k, for simplicity. So, we can calculate $A^(2^p)$, for each p from 0 to $\log_2 x$, in $O(k^3 \cdot \log x)$. With this matrices, we will do a greedy algorithm to determine the minimum number of ski lifts necessary. Note that we consider separately whether it is possible to spend x minutes using 0 or 1 ski lifts, but we won't discuss it as it is very easy.

We will iterate over the powers of 2 of the matrix that we previously calculated, going from $p = \log_2 x$ to $p = 0$, while we keep a current matrix, the current ski lifts we have taken and the answer. Then, for each p, if merging our current matrix with $A^(2^p)$ results in another matrix B such that the maximum element of B (plus taking into consideration the initial path to the ski lift and the final path from the ski lift, which is just small casework) is at least x, then we set the answer to be $2^p +$ the current ski lifts we have taken. Otherwise, we sum 2^p to the current ski lifts we have taken and set the current matrix to matrix B.

Final complexity: $O(k^3 \log x)$