,

Problem A. Elevator

Bob has to go up the stairs $(x-1)$ times to reach floor x. That would take him $20 \cdot (x-1)$ seconds.

An elevator has to move $(y-1)$ floors down, then $(x-1)$ floors up, to reach floor x starting from floor y. That would be $5 \cdot ((y-1)+(x-1))$ seconds.

So the answer is the smaller of these two values.

Problem B. Three Corners

Since the problem statement guaranteed that the rectangle has a positive area and its sides are parallel to the coordinate axes, then exactly two of its vertices lie on one coordinate x and two more on the other (similarly for the coordinates y). Using this fact, we can find the x coordinate of the missing point — the number that occurs exactly once among the values x_1, x_2, and x_3, since there should be 2 points on this coordinate, and now there's only one. In the same way, you can find the y coordinate of the missing point.

Problem C. Chocolate Splitting

Since in each operation you split one part vertically or horizontally, we can prove that each part is always some rectangle of size $x \times y$ where $1 \le x \le n$ and $1 \le y \le m$ (it can be proven by induction). From the other side, we can always create a part $x \times y$ for any $1 \le x \le n$ and $1 \le y \le m$ with no more than two splits.

Let's say without loss of generality that $n \le m$ (otherwise, we can swap n and m). We should find a rectangle with area equal to p or $x \cdot y = p$. It means that x is a divisor of p and y is just $\frac{p}{x}$.

Note that since x is a divisor of p then $\frac{p}{x}$ is also a divisor of p, so it's enough to find only $x \le \frac{p}{x}$. In other words, why should we check pair 5×2 if we already checked pair 2×5.

Finally, since $x \le \frac{p}{x}$ then $x^2 \le p$ and we need to check only $\sqrt{p}$ different x-s. And since $n \le m$, we should only check that p is divisible by x and $x \le n$ and $\frac{p}{x} \le m$.

The total complexity is $O(t\sqrt{p})$.

Problem D. Parking

First, let's try to park a large cars. We will iterate from left to right, and if the current character and the next character are equal to `0` (that is, these places are empty), we will park a large car in the current and next space (if $a > 0$) and reduce a by one. Also replace the current and next character of the string with `1`. If we have checked all the positions, and the value of a is greater than zero, then there is no answer (print `NO`).

After that, we will once again iterate through all the positions and park all the small cars. If the current character of the string is `0` and $b > 0$, we park a small car in this space, and also reduce b by one. If we have checked all the positions, and the value of b is greater than zero, then there is no answer (print `NO`).

Now $a = 0$ and $b = 0$, so we can print `YES` and the assign itself.

Problem E. Pairs of Numbers

Let's consider the process backward. For the pair (x, y) with $x \neq y$, it's easy to find the pair that has produced it: it's either $(x-y, y)$ if $x > y$, or $(x, y-x)$ if $x < y$. So, the naive solution would be to subtract the smaller number from the greater number, until they become equal. If we arrive at the pair $(1, 1)$, then the initial pair will be written on the board, and the number of days is equal to the number of subtractions we made; otherwise, the initial pair won't be written.

Unfortunately, this solution is too slow even for the example tests. How to optimize it? Let's replace the subtraction with modulus operation: after that, we get something similar to the Euclidean algorithm that finds the greatest common divisor of two integers, and we know that it works in $O(\log \min(x, y))$, so if we replace subtractions with modulus, our solution will become fast enough. Be careful though: since

modulus operation is equal to multiple subtractions, we should calculate the number of days we "skipped" using each modulus operation; and the resulting pair will not be a pair of two equal numbers, but a pair of zero and some other number (so, we will model an additional day).

Problem F. Tree Cutting

Let the original distance between adjacent trees be 1. Let's then iterate over the final distance between adjacent trees.

An arrangement of m trees such that the distance between the adjacent ones is d takes $(m-1) \cdot d + 1$ consecutive positions.

Thus, we can also consider all possible ways to choose the first position of that arrangement so that it fits in n positions. That would be $n - ((m-1) \cdot d + 1) + 1$ options.

You might also have to treat the cases where m trees with distance d don't fit at all carefully.

Overall complexity: $O(n)$.

Problem G. Educational Game

Let's calculate dynamic programming $dp_{i,j}$ — the number of ways to read the suffix of a given string $s[j..|s|]$, starting from the j-th cell of the word in the i-th row. If $j = |s|$, then $dp_{i,j} = 1$, otherwise we can choose where the next character will be, i.e. $dp_{i,j} = dp_{i+1,j+1} + dp_{i,j+1}$ (the character in the next row or in the current one).

Then the answer to the problem is the sum over all i of $dp_{i,1}$.

Problem H. Vacation

The first important idea is the following: if the arrangement of 1's and 2's is fixed, then it's optimal to take the cheapest swimsuits of each type.

Consider the most optimal possible answer for some i: the cheapest i swimsuits disregarding the type. If you can arrange them in a valid way, then it's the best answer.

Let there be cnt_1 swimsuits of type 1 and cnt_2 swimsuits of type 2. You can deduce the existence of a valid construction based on some inequalities on cnt_i. WLOG assume that $cnt_1 \geq cnt_2$. Consider the tightest possible construction for 1's: $112112\ldots211$. You can see that it's impossible to have a construction for $cnt_1 > 2 \cdot (cnt_2 + 1)$. Otherwise, you can build a construction for the number of 2's equal to cnt_2 and keep removing 1's from it until there are exactly cnt_1 of them.

So there should be $cnt_1 \leq 2 \cdot (cnt_2 + 1)$ and $cnt_2 \leq 2 \cdot (cnt_1 + 1)$.

However, what should we do if the construction can't be obtained? That means that there are either too many 1's or too many 2's. Let $cnt_1 > cnt_2$ again. Your only option is to keep removing 1's until the construction is doable. By removing a 1 and adding an extra 2 you never decrease the answer, since at the beginning we assumed that the taken i swimsuits are the cheapest ones.

Thus, you can calculate the minimum number of 1's to remove and obtain the answer using some partial sums on each of the type.

Overall complexity: $O(n \log n)$.

Problem I. Cards and Numbers

The easiest solution to this problem handles two different cases: the numbers on the sides of the resulting cards will be either the same or different.

The first case is fairly easy: we should make all numbers equal to each other, so we should find a number that occurs the maximum number of times in the input, and turn all of the numbers into it.

The second case is a bit tricky. If we want to turn all cards into (x, y) or (y, x) (and $x \neq y$), then we should count the number of *cards* that contain x plus the number of *cards* that contain y, not the number

of occurrences of x and y, since the card (x, x) still requires one operation to turn it into (x, y), and the same holds for (y, y). So, the second possible answer is to find two numbers with the maximum number of cards that contain these numbers at least once.

Choosing the minimum between the answer for the first option and the answer for the second option, we obtain the answer to the problem.

Problem J. Balloons

Note that all $b_i \leq 4$ and at each moment only one person can use the inflation station. It means that any person, who last used the station at least 4 minutes ago is now rested and can be appointed again.

In other words, the only persons we should care about is the last 4 users of the station. That's why we can define states as tuples (t, i_1, i_2, i_3, i_4) where t is the current time and i_j are indices of last 4 persons or $n + 1$ if there are no persons who used the station that minute.

The transitions are quite straightforward: you choose any person i_5 who can inflate balloons this minute and move to $(t+1, i_2, i_3, i_4, i_5)$. Since you move from t to $t+1$ you can write dp in two layers if you want.

This solution works in $O(m \cdot n^5)$ time and $O(n^4)$ memory.

The other solution is to store as state the current time and how many minutes each person have yet to rest. There will be $O(t \cdot 5^n)$ states and $O(n)$ transitions. Carefully written it also works fast enough. Moreover, it looks like it's always enough to watch only at 5 persons with maximum a_i, so there will be $O(t \cdot 5^{\min(5,n)})$ states.

Problem K. Rook Coloring

First of all, let's calculate the number of ways to color the rooks into 0, 1, 2, 3, ..., 9 colors such that the order of these colors doesn't matter, and each color is used. There are multiple ways to do this, perhaps the easiest one is to use a recursive function that takes the current position, the number of used colors, and the colors of previous rooks as the arguments, and chooses the color for the next rook (this color is either one of the previously used colors or an entirely new color). The number of possible colorings into at most 9 colors is not very big, so even this basic recursive function will work fast enough (though it can be optimized using, for example, bitmask dynamic programming).

Okay, now we know cnt_i — the number of ways to color the rooks into exactly i colors (order doesn't matter). How to calculate the number of ways to color the rooks into n colors, if their order matters? Let $A(n, i)$ be the number of ways to choose i elements from n (and their order matters). We can see that the answer to the problem is exactly $\sum_{i=0}^{9} A(n, i) cnt_i$, but how to calculate $A(n, i)$ fast enough? When i is small, $A(n, i) = \frac{n!}{(n-i)!}$ can be rewritten as $\prod_{j=0}^{i-1} (n - j)$, and this expression has up to 10 multiples (so it can be easily calculated).

There is another solution to this problem based on chromatic polynomials, though it is much slower in practice.

Problem L. New Road

Instead of calculating the remoteness as the sum of distances between any pair of vertices let's look for each edge how many paths are going through this edge.

Since the given graph is the tree, and we add one edge, then the added edge will create one cycle, and we can look at the whole structure as at the cycle there each vertex is a root of some subtree.

It means there are essentially two types of edges: cycle edges and tree edges. If we look at an edge in any tree then we can note that if we erase this edge, the whole graph will fall apart in two components: a subtree of size x and all other $n - x$ vertices. It means that any path from one component to the other component must contain this edge and only such paths will contains this edge. In other words, for each

tree edge, we should add $w \cdot x(n-x)$ where w is the weight of the edge and x is the size of its subtree.

The cycle edges will be processed in the other manner. Let's find this cycle $c_1, c_2, \ldots, c_k$ and say that sz_i is the size of tree rooted at vertex c_i.

We can note that each path that goes through at least one of the cycle edges will go in the cycle at some vertex c_i and will go out at some other vertex c_j. Since all paths are undirected, we can always say that any such path will start at some vertex c_i and will go clockwise till vertex c_j.

What can we say about this part c_i–c_j? Its length $curLen$ should be less than the half of the cycle length $total$, or $2 \cdot curLen < total$ (case, where $2 \cdot curLen = total$ should be handled separately and carefully). But if $2 \cdot curLen < total$ then any path that starts in c_i-th tree and ends in c_j-th tree will go along this part, so we should add $sz_i \cdot sz_j \cdot curLen$ to the answer.

Also, for each vertex c_i there is such vertex lst_i that any path that starts in c_i and finishes before lst_i has length lower than $total/2$ and all paths that finishes at lst_i or further will have length greater that $total/2$.

Note that $lst_{i+1} \geq lst_i$, so we can try two pointers technique. Let's maintain some information that allows us to calculate the contribution of all paths which starts at c_i. We can write the contribution as formula

$$\sum_{i \leq j < lst_i} sz_i \cdot sz_j \cdot len(i,j) = sz_i \sum_{i \leq j < lst_i} sz_j \cdot len(i,j)$$

where $len(i,j)$ is the length of cycle path from i to j ($len(i,i) = 0$). So. we should maintain valid value $curSum = \sum\limits_{i \leq j < lst_i} sz_j \cdot len(i,j)$ and add $sz_i \cdot curSum$ for each vertex i.

How will $curSum$ change if we move from i to $i+1$? All $len(i,j)$ will change to $len(i+1,j)$, i. e. will decrease by the length of edge between c_i and c_{i+1} (denote it as es_i). In total, $curSum$ will be decreases by $es_i \sum\limits_{i<j<lst_i} sz_j$. So let's maintain the value $curSz = \sum\limits_{i<j<lst_i} sz_j$ along with $curSum$. Recalculating $curSz$ is easy.

When we move the right border j from lst_i to lst_{i+1} we maintain current path length as $curLen$, so $curSum$ will be increased by $sz_j \cdot curLen$ and $curSz$ — by sz_j.

In other words, we can use two pointers to calculate the contribution of all cycle edges in $O(|cycle|)$. In total, for each project we can find the cycle in $O(n)$, calculate the contribution of all tree edges in $O(n)$ and all cycle edges in $O(n)$. So, the result complexity is $O(nm)$.

Problem M. Binary Strings

Let's rephrase the problem as follows: given an undirected graph of n vertices with edges of the form $(v, 2v)$ and $(v, 3v)$. In this graph, we have to count the number of independent sets of vertices (such a set of vertices that no two vertices from the set are connected by an edge). It is easy to see that the graph consists of several connected components. Therefore, for each component, you can calculate the answer, and then multiply them.

Consider a component that contains the vertex 1. In such a component, all vertices will have the form $2^a 3^b$ (for non-negative integers a and b). To solve the problem for such component we can calculate the dynamic programming $dp_{i,mask}$ — the number of independent sets if we considered all the vertices for which $a \leq i$, and among the numbers of the form $2^i 3^b$ we have chosen the values of b in the $mask$ ($mask$ — binary mask of size 12 for our problem). For the transition from i to $i+1$, we can iterate over the new mask $nmask$ and check that there are no two adjacent bits set in it (because these numbers differ by 3), and there is no such bit that it is set in both $mask$ and $nmask$ (because these numbers differ by 2). To speed this part of the solution up, we can use broken profile dp.

Let's look at the component where the minimum vertex is v. Then all the vertices of this component have the form $v2^a 3^b$, i.e. we can divide all the vertex numbers by v and then this will be similar to the problem when the minimum vertex is 1, except that the maximum available number of the form $2^a 3^b$ may differ.

Using this fact, we just need to calculate the answers for the components where all the vertices have the form $2^a 3^b$, and the maximum of them is equal to some K. How many such distinct K exist? Since K is equal to 2 in some power multiplied by 3 in some power, then there is no more than $\log_2 10^6 \log_3 10^6$ such distinct values of K. And for all of them, we can calculate the answer by the dynamic programming described above.

Okay, what about handling queries? Let's calculate the answer for $n = 10^6$. Then, if we want to move from 10^6 to $10^6 - 1$, we can easily find which component contains the vertex we are erasing from the graph, and recalculate the answer for it in $O(1)$ using already calculated values for different K.