Problem A. Katya and the Broken Keyboard

Problem author: Ekaterina Vedernikova, developer: Nikolay Vedernikov

By going through the string, we will maintain $cur[c]$—how many times the letter c has been encountered. There are three possible cases:

1. the letter c is not broken, then we add 1 to the answer,
2. the letter c is broken and the count $cur[c]$ is divisible by $(x[c]-1)$, then the next press will not work, so we add 2 to the answer,
3. the letter c is broken and the count $cur[c]$ is not divisible by $x[c]-1$, then the next press will work, so we add 1 to the answer.

Problem B. Cooperative Game on a Tree

To solve this problem, the method of dynamic programming (more specifically, the method of dynamic programming over subtrees) is proposed to use. Let's define dp_v as follows:

- $dp_v = 0$, if v is a leaf
- otherwise, dp_v equals the answer to the problem if the game starts at vertex v.

Now let's denote by d_a the depth of vertex a, and by $S_{v,d}$ the set of vertices that are in the subtree of vertex v and are at depth d (all depths are considered relative to the entire tree). It is not difficult to see that if the game starts at vertex v, and the first leaf that the red chip reaches is vertex r, then the maximum number of chips that can be obtained in the game is $1 + \max\limits_{b \in S_{v,d_r}} dp_b$ (in this formula, b stands for the vertex where the blue chip will be when the red chip reaches r). Therefore, dp_v will be equal to the maximum value of this expression overr all leaves r in the subtree of v.

Note now that if b_1 is an ancestor (not necessarily a direct one) of b_2, then $dp_{b_1} \geq dp_{b_2}$. Indeed, if there exists a strategy that starting from vertex b_2 can achieve k chips on the tree in the end, then for vertex b_1 the same result can be achieved by moving both chips to b_2 and then using that strategy. From this, it follows that for the answer it is sufficient to consider $1 + \max\limits_{b \in S_{v,dr_v}} dp_b$, where dr_v is the shallowest depth of all the leaves in the subtree of v (since all vertices where the red chip could have been respawn either lie in S_{v,dr_v} or are descendants of some vertex from this set, and then the value of dp for them is not optimal).

In total, to calculate dp_v, it is enough to find the depth dr_v of the shallowest leaf in the subtree of v, and then take the maximum value of dp over all vertices of the subtree of v that are at depth dr_v and add 1 to it. The values of dr_v can be easily calculated for all vertices of the tree with a single depth-first search. Furthermore, during the recalculation of the dynamics for each depth of the original tree, one can maintain an array that lists the values of dp for all vertices at that depth in the order of traversal. Then for any vertex v, S_{v,dr_v} will correspond to a certain subsegment of such array for depth dr_v. The exact boundaries of this subsegment can be found using binary search, for which you will also need to calculate the entry and exit times tin_v and $tout_v$ for each vertex. To quickly find the minimum on a segment, it is necessary to use a data structure, e.g. a segment tree. In this case, the total time for calculating all dynamic values will be $\mathcal{O}(n \log n)$, and the answer to the problem will be dp_1.

Problem D. Count the Christmas Trees

Let's solve a simpler problem: calculate the number of ways to add the n-th layer if we already have $n-1$.

We will add edges from left to right. Then, let's say i edges out of j vertices of the $n-1$ layer have already been drawn. If j is not the last vertex of the $n-1$ layer, then from the $j+1$-th vertex, we can draw from 0 to 2 edges, meaning we can make i, $i+1$, or $i+2$ edges and $j+1$ vertices.

In this case, we will calculate the following two-dimensional dp: $\texttt{dp}[i][j]$, where i is the number of drawn edges, and j is the number of vertices from which the edges have been drawn. The dynamic programming transition will be as follows:

$$\texttt{dp}[i][j] = \texttt{dp}[i][j-1] + \texttt{dp}[i-1][j-1] + \texttt{dp}[i-2][j-1]$$

The answer for such a problem will be found in $\texttt{dp}[n][n-1]$.

Let's solve the original version of the problem: we need to calculate the number of Christmas trees of height n. We know how to count the number of ways to add a new layer to a Christmas tree. Note that the layers are independent, so the answer will be the product of the number of ways to add the $2, 3, \ldots, n$ layers. Through dp, this value is equal to

$$\texttt{dp}[2][1] \cdot \texttt{dp}[3][2] \cdot \ldots \cdot \texttt{dp}[n][n-1]$$

Thus, we can precompute the dp values and output the required product.

Problem E. Not Everything Is So Ambiguous

Let x be the secret number, b the base of the numeral system, and n the number of digits in the base-b representation of the number x.

Then we find by binary search the smallest d_0 such that $x + d$ has more digits than x. Obviously, this will be a number of the form b^n (i.e., the first $n+1$-digit number in the base-b numeral system).

Let's prove that it is enough to take 10^{12} as the upper limit: if $b = 1000$ and $x = 10^9$, then $n = 4$ and the nearest five-digit number is exactly 10^{12}; if $b < 1000$, then since the number $x \cdot b$ will have one more digit than x, then $x \cdot b \le 10^9 \cdot b < 10^9 \cdot 1000 = 10^{12}$. If $b > 1000$, then the number cannot have more than three digits, so $x + d$ will be no more than $b^3 \le 2023^3 < 10^{12}$.

After that, we search by binary search for the smallest D such that $x + d_0 + D$ has more digits than $x + d_0$. This will be b^{n+1} (i.e., the first $n+2$-digit number in the base-b numeral system). Then $b^{n+1} - b^n = D$, knowing n and D, we find b, and then we find x. For the second binary search, it is enough to take 10^{15} as the maximum (the proof is completely analogous to the proof for the previous upper bound).

Thus, the first binary search results in no more than $log_2(10^{12}) < log_2(2^{40}) = 40$ queries, and the second in no more than $log_2(10^{15}) < log_2(2^{50}) = 50$, totaling no more than 90 queries.

Problem F. Magic Square

Problem author and developer: Nikolay Vedernikov

Note that the sum in each row and in each column equals $\frac{n \cdot (n^2+1)}{2}$.

We will call row or column *bad* if its sum is not equal to $\frac{n \cdot (n^2+1)}{2}$.

After two numbers were swapped, at most two columns or two rows could have become *bad*, because the others remained unchanged.

Then there are three cases:

1. Two *bad* columns and zero *bad* rows, then the numbers that Gargamel swapped are in one row. We iterate through the row and check that the numbers at the intersection of the row and the two *bad* columns, when swapped, will make the square magical again.
2. Zero *bad* columns and two *bad* rows, similar to the previous point.
3. Two *bad* columns and two *bad* rows. Then consider the four numbers that are at the intersection of the *bad* rows and *bad* columns. Among them, the numbers located in opposite corners of the resulting rectangle were swapped. We select the two numbers that, when swapped, will make the square magical again.

Problem G. Casino

Author and developer: Alexander Ponkratov

Let's divide all numbers into classes based on the remainder when divided by 3. Denote by cnt_i the number of numbers with remainder i ($0 \le i < 3$). Let's assume that there are no numbers with a remainder equal to 0 for now.

If we replace all numbers with their remainder when divided by 3, then the first element of the sequence uniquely determines the entire sequence. There are only two possible options: $1, 1, 2, 1, 2, 1, 2, \ldots$ and $2, 2, 1, 2, 1, 2, 1, \ldots$. The sequence starts with one of the numbers from the class that has the greater count. The number of ways to arrange them is $cnt_1! \cdot cnt_2!$. Also note that if $cnt_1 = cnt_2$ or the difference between cnt_1 and cnt_2 is more than 2, then there are no suitable cases

Let's return the numbers with a remainder equal to 0 to the array. Such numbers do not change the sum on the prefix modulo 3, so they can be placed in any position except the first. The number of ways to do this is $cnt_0! \cdot C_{n-1}^{cnt_0}$.

The final answer: $cnt_1! \cdot cnt_2! \cdot cnt_0! \cdot C_{n-1}^{cnt_0}$.

Problem H. Scooter Numbers

Problem author: Konstantin Kokhas, developer: Rita Sablina

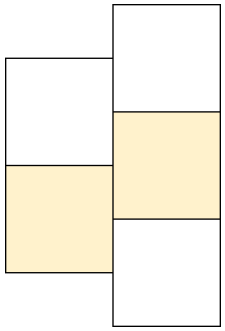
Let d_{nk} be the number of partitions of the number n into summands such that $mex \ge k$. In such partitions, numbers from 1 to $k-1$ must be present.

For $k = 1$, in d_{n1} each partition with $mex = 1$ occurs once. For $k = 2$, in d_{n1} each partition with $mex = 2$ occurs once, and in d_{n2} each partition with $mex = 2$ occurs once. For $k = p$, each partition with $mex = p$ occurs once from d_{n1} to d_{np}. Therefore, if we sum $\sum_{i=1}^{p} dp_{ni}$, each value of i will repeat in the sum as many times as there are partitions with $mex = i$.

Counting can be done using dynamic programming. $d[i][k]$ is the number of partitions of the number i into summands, with the maximum number in the partition $\le k$. The recurrence formula is $d[i][k] = d[i-k][k] + d[i][k-1]$. The desired sum for all $k \ge 1$, $n - \frac{k \cdot (k-1)}{2} \ge 0$ is $\sum_k dp[n - \frac{k \cdot (k-1)}{2}][n]$—the partition with $mex = k$ contains all numbers from 1 to $k-1$, so for each k, we need to sum the partitions of the number $n - \frac{k \cdot (k-1)}{2}$ with the maximum number in the partition $\le n$.

Problem I. Squares

First, we need to understand what a *covering* looks like. Four simple options are obtained if all the squares are arranged in a regular grid. Otherwise, there will be two squares that do not touch evenly, for example, as shown in the picture:



If there is a pair of squares, as in the picture, then in the two columns where these squares are located, all the other squares must go sequentially, adjacent to each other, with a step of 2 units. Thus, a column

was found that is filled with columns. Reasoning in this way, it can be noticed that all the squares from the *covering* are divided into columns. It is easy to understand that in other cases, the *covering* will be divided into rows or columns covered with squares.

So, our entire infinite table is divided into rows or columns in 4 ways (we choose what the table is divided into: rows or columns, and in each of the two selected options, we also need to choose which columns/rows will be encountered in the *covering*: even or odd). We will be looking for the largest *good* subset, which complements each of the four *covering* options independently.

If the table is divided into columns, then for each of the columns, we can maintain the number of squares that occur in it at even and odd heights. Then it is clear that in the largest *good* subset in this case, and in this column, the largest number of the two calculated will be encountered. The problem is then solved using a simple technique and is solved in $O(n \log n)$ if `std::map` is used and is solved in $O(n)$ if a hash table is used.

Problem J. Streets of Flatland

Problem author and developer: Daniil Oreshnikov

The first fact to note is that the number of highways is exactly equal to the number of roads $n-1$ minus the number of pairs of compatible roads that we place in one highway. Indeed, each action of the type "combine the highway ending in road (u, v) and the highway starting in road (v, w) into one" reduces the number of highways by 1. We will sequentially select pairs of compatible roads and merge their highways — then the number of highways will be $(n-1)$ − `<number of actions>`, and each action corresponds to a pair of merged roads.

Thus, we have reduced the problem to finding the maximum number of pairs of roads that can be combined according to the rules described in the statement. Moreover, this problem can be solved independently in the vicinity of each vertex v: if the roads (a, b), (b, c), and (c, d) form one highway, then with a fixed (b, c), the choice of a does not depend on the choice of d. In other words, it is enough for each vertex v to find the maximum matching on the roads emanating from it.

To find the maximum matching on the roads emanating from vertex v, it should be immediately noted that each road has no more than two compatible with it. Indeed, the definition of compatibility in simple terms is formulated as "the vector $\overrightarrow{vw}$ must be the closest to $\overrightarrow{uv}$ when rotated around v". The closest can only be one when rotating clockwise and one — counter-clockwise.

Now simply consider each vertex v and the roads around it separately. Build a graph on these roads, where two roads will be connected by an edge if they are compatible. To do this, it is enough to sort the vectors of the roads by their angle, after which either use the method of two pointers or search for neighboring roads by angle for each vector $\overrightarrow{uv}$ using binary search. After that, it remains only to find the maximum matching in the resulting graph. But in it, the degree of each vertex does not exceed 2, which means the graph is divided into paths and cycles. In such a graph, the size of the maximum matching in each connected component of size x is equal to $\left\lfloor \frac{x}{2} \right\rfloor$.

The total runtime of such a solution is $\mathcal{O}(n \log n)$.

Problem K. Guess the String

Let's introduce two strategies for guessing a string: F (if nothing is known about the string) and D (if it is known that the first symbol of the string is equal to `b` or `c`). The maximum number of queries for a string of length n, if we use the corresponding strategy, will be denoted as F_n and D_n.

- $n = 0$. The fictitious states F and D require 0 actions each;
- $n = 1$. We understand that there is another symbol to the left of the last one, which we already know. Accordingly, in strategy F, we need to make 2 queries, applying the suffix of the string `ab` and `ac`, and in strategy D only one query `ab` is sufficient. That is, $F_1 = 2$ and $D_1 = 1$.

- $n \geq 2$, strategy F. Let's make a couple of queries `aa` and `ba`, attaching these strings to the beginning of the string. Based on the answers to these queries, we uniquely restore the first symbol of the string and understand whether the second symbol is `a`, thus $F_n = 2 + \max\{F_{n-2}, D_{n-1}\}$.
- $n \geq 2$, strategy D. Make a query `ba`.
  - Received 0, then we restored the first symbol and can move on to strategy D_{n-1};
  - Received 1, then the following options for the first two symbols of the string are possible: ca, bc, bb, make a query `bb` and restore the first two symbols of the string, moving on to strategy F_{n-2}.
  - Received 2, then we can move on to strategy F_{n-2}.

Thus, $F_n = 2 + \max\{F_{n-2}, D_{n-1}\}$ and $D_n = \max\{2 + F_{n-2}, 1 + D_{n-1}\}$. But considering that $F_1 \leq F_2 \leq F_3 \leq \ldots$ we can conclude that:

$$F_n = 2 + \max\{F_{n-2}, D_{n-1}\} = \max\{2 + F_{n-2}, 4 + F_{n-3}, 3 + D_{n-2}\} = \\ = \max\{2 + F_{n-2}, 4 + F_{n-3}\}, n \geq 3$$

We get $(F_0, F_1, F_2) = (0, 2, 3)$ and $F_n = 4 + F_{n-3}$, from which it is easy to understand that $F_n < 4/3n + 1$, which means $F_n = \lceil \frac{4}{3} n \rceil$.

Problem L. a, ab, ba Strings

Author: Alexander Chistyakov, developer: Alexander Ponkratov

Let's split the string into substrings that start with '`b`', end with '`b`', and the characters before and after are also equal to '`b`'. Among all substrings, one can choose the substrings of the minimum length that, when combined, give the original string. Each such substring can be correctly split if and only if the number of '`b`' characters does not exceed the number of '`a`' characters. Then the task is reduced to maintaining such substrings and checking the correctness of the split. This can be done by keeping all substrings in a `std::set` and storing the indivisible ones in a separate set; when responding to a query, check that the query segment does not contain indivisible substrings; when changing a character, check that some substrings could have merged into one or split into several. The number of characters on a segment can be counted using a Fenwick tree.

There are also alternative solutions that use a segment tree.

Problem M. Three Suitcases

Problem author and developer: Ekaterina Vedernikova

To solve the problem, let's consider 5 cases of sending the suitcases:

1. Check in each suitcase separately;
2. Check in the 1st and 2nd suitcases together, and the 3rd separately;
3. Check in the 1st and 3rd suitcases together, and the 2nd separately;
4. Check in the 2nd and 3rd suitcases together, and the 1st separately;
5. Check in all three suitcases together.

In each case, calculate how much you need to pay for the checks. And choose the case where we pay less money.