

Let's assume that empty arrays don't exist. Then, for each character '{', '}', ',' and letter of the English alphabet, we can define a set of commands to execute:

1. Character '{':
 - ☐ Print '{' and a newline
 - ☐ Increase current indentation by 2
 - ☐ Indent
2. Character '}':
 - ☐ Print a newline
 - ☐ Decrease current indentation by 2
 - ☐ Indent
 - ☐ Print '}'
3. Character ',':
 - ☐ Print ',' and a newline
 - ☐ Indent
4. Letter **c**
 - ☐ Print **c**

Additionally, we'd need to check the case when an array is empty and equal to "{}".

Necessary skills: if-else, strings

Category: ad-hoc, parsing

Task PROZOR	Author: Ivan Paljak
--------------------	----------------------------

The limitations from the task allow us to evaluate every possible shot. In other words, we can assume for each pixel that it is the one located in the upper left corner of the racket and then count how many flies would be affected by that shot. When done with this procedure for each pixel, we know the position of the upper left pixel of the optimal shot, and that is sufficient in order to reconstruct the image and score all points for this task. Additionally, we need to make sure that the racket is located entirely within the image and that we won't affect the flies located on the edges of the racket.

The time complexity of the aforementioned procedure is $O(R^2 \cdot S^2)$. For implementation details, consult the official solution. Moreover, we encourage you to come up with a solution of this task given the limitation ($3 \leq R, S \leq 1000$).

Necessary skills: matrices

Category: ad-hoc

Task OZLJEDA	Author: Ivan Paljak
---------------------	----------------------------

Firstly, let's notice that the first $k+1$ elements of the xorbonacci sequence are cyclically repeated throughout the entire sequence. In other words, it holds $X_A = X_{A+k+1}$ for every A . The proof of this statement is left as an exercise to the reader.

Now we can recall some properties of the xor operation (labeled with \wedge). More precisely, we are interested in the following properties:

$$\begin{aligned}A \wedge 0 &= A \\A \wedge A &= 0 \\A \wedge B &= B \wedge A\end{aligned}$$

Using these properties, we deduce that it holds:

$$X_L \wedge X_{L+1} \wedge \dots \wedge X_R = (X_1 \wedge X_2 \wedge \dots \wedge X_R) \wedge (X_1 \wedge X_2 \wedge \dots \wedge X_{L-1})$$

Using the sequence's cyclicity from the first paragraph and the aforementioned properties, it is clear that the xor of the first N elements of the xorbonacci sequence is equal to the xor of the first N' elements of the xorbonacci sequence where $0 \leq N' \leq 2(K+1)$. Therefore, it is enough to preprocess the xors of the first $2(K+1)$ elements and, using them, answer all given queries in $O(1)$.

Unveiling the explicit relation between N and N' is left to you as an exercise. If you don't succeed, consult the official source code.

Necessary skills: properties of xor, prefix sums (xors)

Kategorija: ad-hoc

Task OTPOR	Author: Branimir Filipović
-------------------	-----------------------------------

Firstly, we must notice that each circuit \mathbf{S} is either in the form of $(\mathbf{S}_1|\mathbf{S}_2|\mathbf{S}_3|\dots|\mathbf{S}_n)$ or in the form $(\mathbf{S}_1-\mathbf{S}_2-\mathbf{S}_3-\dots-\mathbf{S}_n)$, where each of $\mathbf{S}_1, \mathbf{S}_2, \mathbf{S}_3, \dots, \mathbf{S}_n$ is either in one of the forms of \mathbf{S} or equivalent to some form of $\mathbf{R}_1, \mathbf{R}_2, \mathbf{R}_3, \dots, \mathbf{R}_m$.

Therefore, we can write the following pseudo code for the recursive function that solves the task:

res(\mathbf{S}):

 if $\mathbf{S} = \mathbf{R}_i$ then return \mathbf{R}_i

 else if $\mathbf{S} = (\mathbf{S}_1-\mathbf{S}_2-\mathbf{S}_3-\dots-\mathbf{S}_n)$ then return sum(res(\mathbf{S}_i))

 else return $1/(\text{sum}(1/\text{res}(\mathbf{S}_i)))$

Necessary skills: strings, recursion

Category: ad-hoc, recursion, parsing

Task PROSTI	Author: Mislav Balunović
--------------------	---------------------------------

Let's assume a fixed **K** (the number of consecutive numbers) and denote the number of happy numbers in the set $\{i, i + 1, \dots, i + K - 1\}$ with $f(i)$. It is clear that $|f(i) - f(i + 1)| \leq 1$.

Additionally, we can use brute force to find the first 150 consecutive composite numbers. If they begin with the number **j**, then it will hold $f(j) = 0$ for each $K \leq 150$.

Lemma: Let a and b be integers and let $f(a) \leq L \leq f(b)$. Then there exists x from the interval $[a, b]$ such that $f(x) = L$.

The proof of this lemma is simple and is left as an exercise to the reader.

Now, let's work with the numbers **K**, **L**, **M** from the task.

By applying the lemma to $a = j$ and $b = 1$, we know that a solution must exist in the interval $[1, j]$. However, iterating over the entire interval would be too slow.

We use the binary search technique. Let's assume that we know for sure that our solution is in the interval $[a, b]$. Let $c = (a + b) / 2$.

Then we can apply the lemma to one of the intervals $[a, c]$ or $[c, b]$ and solve the task recursively.

The total time complexity of this algorithm is $O(Q * \log j)$.

Necessary skills: combinatorics, binary search

Category: mathematics

The task is to find the longest magical subarray of a subarray from the original array. An array is magical if all the value of all elements are between the values of the first and last element of that array. The solution for 70% of total points has a complexity of $O(N \sqrt{N})$ and will not be explained here, but you can consult the official solution for it. Regarding the solution for all points, special thanks to Mislav Bradač for coming up with such solution and implementing it.

To begin with, we will solve the case when the first element is the minimal element in the magical subarray, and the last element is the maximal. The case when the first element is maximal and the last minimal is solved analogously over an array with all elements replaced with their negative values.

Let's sort all queries with regards to the right end and now sweep from left to right, adding the elements and processing the queries. The idea is to, while sweeping and located at position p , maintain the array $A[x]$, such that $A[x]$ = longest subarray whose left end is at position x , and the right end wherever between $[x, p]$. If we have such an array, it is easy to see that, at position p , the answer for query $[L, p]$ is the maximal value $A[x]$, for x from the interval $[L, p]$. Let's try to efficiently maintain this array. The structure that arises as a solution to this kind of problem is usually the tournament structure. It is necessary to update some positions when we add a new value x at position p . Firstly, given the fact that we are solving the first case when the last element is maximal, and the first minimal, it is obvious that the value x will only be important until the first left position where the value $v > x$ is located. We will label this position as l . The details of finding such position is left for later. Additionally, all larger values to the left of x need to be labeled as unusable because they could never be the left end of the interval that passes over x because x is smaller than them. We will describe the details of finding such positions later, also. For the remaining positions from the interval $[L, x]$, we will label the potentially rightmost end as x .

Let's now look at all the operations our tournament structure must support:

- It needs to be able to exclude a position x as the left end. In other words, that it will not be possible in the future for it to be the left position for a value to the right.
- It needs to be able to set the rightmost position that can be the end of some interval to a new value x .
- It needs to be able to determine the current maximal value in an interval.

Therefore, the entire idea of the tournament is it to store for each position x , until the current position p , what the rightmost position y was while x was still alive, or the distance between y and x . All these operations are possible to achieve using propagation. The tournament will store the leftmost alive element from the interval in a node, the current rightmost position for that interval and the largest distance between the left and right position of the magical subarray. The propagation operations of these values are not difficult to derive, and we leave that as an exercise to the reader. What we still haven't covered is finding the first larger value and removing all larger values as unusable for the left end from the tournament. We do this by maintaining two stacks while moving in the sweep, one to search for the first smaller number to the left, the other to search for the first larger number to the left. All numbers that are removed while searching the first smaller number to the left will be exactly the ones we need to kill in the tournament are not anymore possible left ends. The total complexity of the solution is $O((Q + N) \lg N)$.

Necessary skills: sweep, monotonous stack

Category: data structures, stack