Task YODA	**Author: Nikola Dmitrović**

In 30% of test cases, the numbers **N** and **M** were composed of three digits, so the task could be solved by determining all the digits, comparing them and creating new numbers. The task could also be solved by observing the numbers as strings. Here we will observe the numbers as numbers.
The solution is a combination of the algorithm for determining the digits of a number and the algorithm for creating a number. Since we don't want to think about the number of digits, we can assume that there are at most 10 of them, but we can also use a conditional loop. There was one trick in the task. The solution that outputs "YODA" in the case when the new value of the number is equal to zero is not a good solution because there are cases when the number is zero, and "YODA" shouldn't be output (i.e. when the zeroes appear in the same position of digits in the number). Let us notice that the given word will be output only when the new number hasn't been modified in the program.

Solution: (written in Python 3.x)

```
N = int(input()); M = int(input())

newN = newM = 0   # new numbers we're creating
powN = powM = 1   # the powers we use when creating the numbers
flagN = flagM = 0 # the flag to keep track if the number has been
modified

for i in range(10):
      znN = N % 10; N = N // 10
      znM = M % 10; M = M // 10

      if znM >= znN and M > 0:
            newM = znM * powM + newM
            powM = powM * 10
            flagM = 1 # there has been a change in number newM

      if znN >= znM and N > 0:
            newN = znN * powN + newN
            powN = powN * 10
            flagN = 1 # there has been a change in number newN

if flagN == 1: print(newN)
else: print("YODA")
```

```
if flagM == 1: print(newM)
else: print("YODA")
```

Necessary skills: loop, digit-searching algorithm, digit-creating algorithm
Category: ad-hoc

Task HAN	**Author: Ivan Paljak**

If we simulate the process described in the task by constructing a word that consists of the letters Dominik is saying, and for each **UPIT** command we count the required letters in the word, we are left with an algorithm of time complexity O(Q*N), which is sufficient for 40% of total points. An additional 20% of points could have been won by keeping track of how many of which letter appeared so far while constructing the word. Now it is possible to answer each **UPIT** command in O(1).

Finally, in order to completely solve the task, we must notice that, if undisturbed with a command, after first N%26 spoken letters, Dominik will say each letter exactly N/26 times (he will be stuck in a cycle of length 26), where % is the modulo operator, and / is the integer division operator. Using this property, we can simulate each command in constant time. For implementation details, consult the official solution.

Necessary skills: strings, cycles, modular arithmetics
Category: ad-hoc

Task DEATHSTAR	**Author: Dominik Gleich**

It is necessary to notice that the construction $\mathbf{a_i}$ = bitwise or of all $\mathbf{m_{ij}}$ for each **j** is sufficient to meet all the requirements from the matrix. The proof of this fact is left as an exercise to the reader.

Necessary skills: bitwise operations
Category: ad-hoc

Task CHEWBACCA	**Author: Antonio Jurić**

The simplest solution is to construct an entire graph in memory and for each query run a DFS traversal of the tree that will calculate the distance between two given nodes. A DFS traversal will correctly calculate the shortest path because the given graph is a tree, which means this path is the only path between these two nodes. The complexity of this solution is O(**Q** * **N**).

A better solution also constructs the graph in memory, but before answering the queries it calculates the table of the lowest common ancestor. Using this data we can easily calculate the distance between two given nodes:

depth(**x**) - depth(lca(**x**, **y**)) + depth(**y**) - depth(lca(**x**, **y**)), ie. :

-2*depth(lca(**x**, **y**)) + (depth(**x**) + depth(**y**)).

The complexity of this solution is O(**Q** * log[**k**](**N**)).

For the maximal possible **N** from the task, the previous solution wouldn't work because of the too big space complexity: the table of the lowest common ancestor is too large. But, given the regular structure of the tree, we don't need to store the table of the lowest common ancestor in memory, because that information can be calculated on the spot. Let us enumerate the nodes from **0**. Then the i^{th} child of node **x** can be calculated using the formula:

x*K** + 1 + **i**, **i** ∈ [0, **K**-1],

and the father of node **x**:

floor((**x** - 1) / **K**),

where **K** is the order of the tree. These formulas enable us to efficiently calculate the lowest common ancestor using binary search. The time complexity of this solution is O(**Q** * $\log^2$[**k**](**N**)), whereas the space complexity is constant. In the source codes there is a somewhat simpler solution with complexity O(**Q** * log[**k**](**N**)) that does not use binary search.

Necessary skills: DFS, lowest common ancestor (LCA), binary search, mathematical analysis

Category: graph theory

Task GALAKSIJA	**Author: Mislav Balunović**

Let us choose an arbitrary node **r** to be the root of some tree. Let us denote with $\mathbf{p_x}$ the total XOR of all the curiosities on the path from node **r** to node **x**. It can be easily seen that a pair of planets **x**, **y** is boring if and only if it holds $\mathbf{p_x}$ XOR $\mathbf{p_y}$ = 0 ⇔ $\mathbf{p_x} = \mathbf{p_y}$.

Let us notice that we can "reverse" the input data so, instead of destroying the paths, we can add them.

For connecting the components we will use the union find algorithm where we will, for each component, remember its root, size and, additionally, a hash map that remembers the number of times $\mathbf{p_x}$ appears in the component.
When we need to connect two components, the newly created component is rooted in the root of the larger component. Now we must traverse the entire smaller component (i.e. using the DFS algorithm) and correct the values in the larger component's hash map.

The time complexity of this algorithm is $O(N \lg^2 N)$.

Necessary skills: union find, dfs
Category: graph theory

Task ENDOR	**Authors: Ivan Paljak, Mislav Balunović**

Let us observe two consecutive chameleons moving to the right and are located at coordinates $\mathbf{x_1}$ and $\mathbf{x_2}$, and a chameleon at the coordinate $\mathbf{x_L}$ moving to the left. Between the collision with two chameleons moving to the right, he will pass
$(\mathbf{x_2} + \mathbf{x_L}) / 2 - (\mathbf{x_1} + \mathbf{x_L}) / 2 = (\mathbf{x_2} - \mathbf{x_1}) / 2$ meters.
We can see that the distance traveled doesn't depend on the coordinate of the chameleon moving to the left.

The task is further solved using dynamic programming.
Let f(**i**, **c**) be an array of length **K** that denotes the distance traveled in each color a chameleon colored in **c** moving to the left will take in consecutive collisions if it had just collided with a chameleon **i** moving to the right.
Using the aforementioned statement, we can easily calculate the value of function f.

Finally, for each chameleon moving to the left, we calculate:

1) distance traveled until the first collision;
2) distance traveled between consecutive collisions (calculated using f);
3) distance traveled from the last collision to the end

The time complexity of this algorithm is O(N * K^2).
For implementation details, consult the official solution.

Necessary skills: mathematics, dynamic programming
Category: dynamic programming