CROATIAN OPEN COMPETITION IN INFORMATICS

6th round, February 7th, 2015

solutions

Task PAPRIKA	**Author:** Ivan Paljak

We need to simulate the procedure described in the task. Therefore, we first simulate the swap of ID cards between the first and the second pepper, then between the second and the third, and so on until we reach the end.

When we’ve done all the swaps, we need to count all the peppers that are going to be filled and have an ID card with a number smaller than or equal to **X**, or all peppers that want to be served fresh and have an ID card with a number strictly larger than **X**.

The time complexity of this algorithm is **O(N)**.

Necessary skills: simulation, loops, arrays
Category: ad-hoc

Task NIKO	**Author: Marin Tomić**

Let us denote the set of players that can only play defense by **o**, the set of players that can only play midfield positions by **v**, and the set of players that can only play offense by **n**.
Let us denote the set of players that can play both defense and offense by **on**. Analogously, we have **ov** and **vn**.
Finally, let us denote the set of players that can play all three lines by **ovn**.

For a given formation O-V-N, it is necessary to carefully come up with a layout of the players. We will obviously place the players from sets **o**, **v** and **n** on the lines they can play in. The problem arises when we don't have enough defensive players and need to decide whether to take players that can play both defense and midfield positions or those that can play both defense and offense.

As it usually is with programming, we don't need to make a decision, we can try out all possible combinations and see which one is best.

Let **a** be the number of players from set **on** we will put into defense. Then **on** - **a** players from that set will go into offense.
Let **b** be the number of players from set **vn** we will put into midfield positions. Then **vn** - **b** players from that set will go into offense.
Let **c** be the number of players from set **ov** we will put into offense. Then **ov** - **c** players from that set will go into midfield positions.

Now we are missing only O-**o**-**a**-**c** players in defense, V-**b**-(**ov**-**c**) players in midfield positions and N-(**on**-**a**)-(**vn**-**b**) players in offense. Therefore, the layout is possible only if **ovn** is larger than or equal to the sum of those three numbers.

Now it is enough to try all possible combinations for numbers **a**, **b** and **c** and output "DA" if a layout works out, and "NO" if there isn't a possible layout.

Necessary skills: combinatory thinking, for loop
Category: ad-hoc

Task METEOR	Author: Adrian Satja Kurdija

We need to calculate the shift of the meteor in relation to its initial position or, in other words, the number of fields it fell towards the ground. If we think about it, we conclude that this shift is equal to the minimal vertical distance from a part of the meteor to a part of the ground, minus one. This distance is calculated in a way that we find the lowest part of the meteor and the highest part of the ground for each column; their vertical difference is a candidate for the minimal distance.

After we have calculated the shift, every part of the meteor needs to be shifted. It is most easily done by using a new photograph (matrix of characters) in which we first copy all parts of the ground, fill the rest out with dots, and then add the meteor parts in the calculated positions.

Necessary skills: strings, matrices
Category: ad-hoc

Task KRATKI	**Author: Dominik Gleich**

The key to solving this task is noticing that in any sequence of length N there has to be a monotone subsequence of length at least sqrt(N). Why is that so?

Let’s try and prove it.

Let’s imagine that we have a set of stacks stacked from left to right, with the feature that the numbers are ascending as we are getting to the top of the stack.
Now we do the following algorithm: go through the numbers in the array from left to right, pushing each number to the top of the first stack that we can from the left so that it still holds that the numbers on all stacks are ascending.

Let’s observe the first case: after this algorithm, we have less than or equal to sqrt(N) nonempty stacks. In that case, there has to be a stack with more than or equal to sqrt(N) numbers on it, and it clearly follows that the number of total numbers on stacks is precisely N. In this case, we have shown that at least an ascending subsequence is of the length sqrt(N).

Let’s now observe the second case: after this algorithm, we have more than sqrt(N) nonempty stacks. Let’s observe the element on top of each stack. We notice that the numbers on top of the stacks are descending, looking from left to right. In the case that they are not descending, it would mean that we hadn’t complied to our construction features during the execution of this algorithm, because we pushed to the top of the first stack that we could so that the next number is larger than the last on the stack, and it is the first stack that we could push to. Having that in mind, we have a descending subsequence of length at least sqrt(N) that consists of numbers on top of all stacks.

We have completed the proof. There is always a monotone subsequence of length at least sqrt(N).

After this, it is fairly simple to see that the next construction gives a monotone sequence of required length K, assuming that K >= sqrt(N), otherwise such a sequence doesn’t exist.

Let the sequence be in the form of [n - k + 1, n] [n - 2k + 1, n - k] [n - 3k + 1, n - 2k], etc.

Each subsequence of length K has an ascending subsequence of length K, and because there are <= K such sequences, because K >= sqrt(N), a descending sequence of length larger than K does not exist.

Necessary skills: for loop
Category: ad-hoc

Task NEO	**Author:** Ivan Paljak

It is crucial to see that the following statement holds:
Matrix A is extremely cool if and only if every single one of its submatrices of dimensions 2x2 is cool.

Accordingly, we can construct a new matrix where the element located at `B[i][j]` is set to 1 if `A[i][j] + A[i+1][j+1] <= A[i][j+1] + A[i+1][j]`. Solving this task now comes down to searching for a maximum area rectangle in matrix **B** that consists of only 1s. We believe that this problem is quite familiar to the contestants, so we will not go into further analysis. Depending on the implementation complexity, it was possible to score 40%, 60% or 100% of total points for time complexities O(n^4), O(n^3), O(n^2), respectively.

We will prove the aforementioned statement using mathematical induction. First, let's write down the definition of a cool matrix as **A**(1, 1) - **A**(1, s) <= **A**(r, 1) - **A**(r, s).

We prove the weaker statement:
Matrix **A** of dimensions 2x**s** is extremely cool if every single one of its submatrices of dimensions 2x2 is cool.

We conduct the induction on s.
Base: s = 2, it holds because every 2x2 cool matrix is extremely cool.
Step: We assume that there is an s such that the statement holds. Let's notice what happens then with matrix of dimensions 2x(s+1).

Since every 2x2 submatrix is cool if it holds **A**(1, s) - **A**(1, s + 1) <= **A**(r, s) - **A**(r, s + 1), while from the step presumption we know that **A**(1, s') - **A**(1, s) <= **A**(r, s') - **A**(r, s) for each s' < s. If we sum these inequalities, we get **A**(1, s') - **A**(1, s + 1) <= **A**(r, s') - **A**(r, s + 1) for each s' <= s.

We conclude that if the statement holds for a matrix of dimensions 2xs, then it must hold for a matrix of dimensions 2x(s+1). In other words, by using the principle of mathematical induction, we have proved that the statement holds for every s >= 2.

Analogously, we can conduct this proof on rows in a way that we fixate two columns and conduct the induction on rows. The formal continuation of the proof is left as an exercise to the reader.

Necessary skills: mathematical induction, finding the largest rectangle in a matrix
Category: mathematics, ad-hoc

Task WTF	**Author:** Adrian Satja Kurdija

The algorithm described in the task seems nonsensical at first glance. Therefore, we need to uncover it and make it more sensible; we do this with a series of clever observations.

First, let's notice that, regardless of the value of constant R, after N rotations the array is restored to its initial state. That means that the second part of the algorithm begins operating on the same array as the first part of the algorithm, except that the signs are changed. However, instead of changing the signs, in the second part of the algorithm we can reduce the sum (instead of increase) by `A[index]`. Effectively, the first and second part of the algorithm operate on identical arrays.

The next step is ignoring the rotation. Instead of imagining an array **A** that rotates, we can imagine **N** different arrays (calculated by rotating array **A**) which the first and second part of the algorithm operate on. On the **i^th^** of these arrays (let's call it **a**), the first and second part of the algorithm jointly perform the following:

```
sum += a[id1] - a[id2 + 1],
```

where `id1` is the smaller, and `id2` is the larger of the two indices `ID[i], ID[i+1]`. The upper difference reminds us of the difference of prefix-sums, except that the difference needs to be reversed. By changing the signs of all elements of array **a**, we get a more familiar relation:

```
sum += a[id2 + 1] - a[id1].
```

If array **a** contains prefix-sums[1] of another array **b**, then the difference of prefix-sums is the sum of the corresponding interval of array **b**. More precisely,

```
a[id2 + 1] - a[id1] = b[id1] + ... + b[id2].
```

The elements of array **b** for which the upper formula holds are calculated using a simple formula `b[i] = a[i + 1] - a[i]`. Notice that array **b** has N - 1 elements.

In a nutshell: the given algorithm actually adds up intervals in arrays $\mathbf{b}_1$, …, $\mathbf{b}_N$ which we can construct. But not just any intervals; the intervals in adjacent arrays are

`[ID[i], ID[i+1]]` and `[ID[i+1], ID[i+2]]`

(after we properly orient them, from smaller to larger boundary). These intervals have **a common edge** (`ID[i+1]`). If we imagine arrays $\mathbf{b}_1$, …, $\mathbf{b}_N$ as rows of a matrix, it follows that the algorithm in fact adds up the **path** in that matrix that goes through the interval from **ID**[1] to **ID**[2] in the first row, goes down to the second row, in it goes through the interval

[1] The k^th^ prefix-sum of array **b** is equal to the sum of the first K - 1 elements from **b**.

from **ID**[2] to **ID**[3], goes down to the third row, in it goes through the interval from **ID**[3] to **ID**[4] and so on until the N^{th} row, all the while adding up all the elements that it visits in the variable `sum`.

Therefore, the task comes down to finding the optimal path in a matrix that goes through an interval from left to right or from right to left in each row, goes down to the following row and continues. This is the actual task, all the rest was a camouflage.

This task is solved using dynamic programming. The state is a position (row, column) where we are located and the previous step of the path (left, right or down). If the previous step was left, the next step cannot be right and vice-versa. You can look at the attached source code for details, or work them out as an exercise.

Necessary skills: unveiling the task, dynamic programming
Category: dynamic programming