CROATIAN OPEN COMPETITION IN INFORMATICS 2012/2013

ROUND 4

SOLUTIONS

COCI 2012/2013	Task OREHNJAČA
4th round, January 19th, 2012	**Author:** Nikola Dmitrović

Each spectator thought he would get **K** - **P** + 1 pieces of walnut roll (that is how many pieces there are in total, marked from **P** to **K**). The solution to the first part of the task is therefore determining the maximum value of **K** - **P** + 1 which we calculate for each spectator individually.

As spectators were taking pieces of the wallnut roll, at some point it might have occured that a spectator could not take a piece that was provided for them because someone else had taken it already. We therefore create an array of a maximum lenght of 1000 where we initialize every element to the value 1 (every piece is in its place). For each spectator marked with **i** we sum up the values in the array marked from P_i to K_i; also, in that interval we change every value 1 to 0 (the piece is not there anymore). During this procedure we keep the maximum value of the sum.

Necessary skills: loop, array

Category: ad hoc

COCI 2012/2013	Task ESEJ
4th round, January 19th, 2012	**Author:** Ivan Mandura

Let us describe the procedure of determining if a word is nice or not. We iterate along the word from left to right and keep the array of letters which we have not been able to pair yet. Assume we have encountered a letter A. We will check if the last unpaired letter is A: if it is, we can pair it with the new letter and remove it from the array of unpaired letters. If the last unpaired letter is B, we must add the new letter A to the array of unpaired letters (we cannot pair it with some previous A because if we try to connect the B afterwards, its arch will intersect with the arch from letter A).

Notice that the array of unpaired letters we keep is a stack: a structure which enables adding elements to the end, having access to the last element and removing the last element. If the stack is emtpy after we are done iterating, the word is nice.

Necessary skills: strings, stack

Category: ad hoc

COCI 2012/2013	Task VOYAGER
4^{th} round, January 19^{th}, 2012	**Author:** Nikola Dmitrović

The solution to this task is an implementation of the given conditions. Let us describe some characteristics of the solution.

At the start, the probe is located in the given position. We have to send the signal in all four directions (up, right, down, left). We must keep the current position and the current direction. Once the signal encounters a planet, it changes its direction.

We keep the current direction as a value marked **d** (**d**=0, up; **d**=1, right; **d**=2, down; **d**=3, left). Let us create two arrays of lenght 4 like this: **RS**[] = {-1, 0, 1, 0} and **CS**[] = {0, 1, 0, -1}. Notice that **RS**[**d**] is the shift in rows, and **CS**[**d**] is the shift in columns.

After we check the value in the current cell, we might have to do a direction correction. We repeat this procedure until the signal leaves the system or until it encounters a black hole. We perform a direction change using bitwise XOR, like this:

if planet = '\' then
   d = d XOR 3
if planet = '/' then
   d = d XOR 1

It is easy to check that the above described direction change is correct.

The additional problem is determining a situation in which the signal enters a cycle. We do that by counting the cells we have already visited. Given that a signal can travel through a cell in four directions, and the system has **N * M** cells, there are **4** * **N** * **M** positions in which the signal can be. When a signal visits more cells than that, it is obviously in a position where it had been before. The next position depends on the current position only, so the signal is in a cycle.

Necessary skills: 2D matrix
Category: simulation

COCI 2012/2013	Task RAZLIKA
4th round, January 19th, 2012	**Author:** Goran Gašić

For the sake of simplicity, let us determine **N** - **K** numbers which will remain in the array. This way we actually determine which **K** numbers should be removed.

Notice that there is always an optimal solution in which we choose **N** – **K** consecutive numbers in a sorted array.

Proof. Let **a** and **b** be the values of the greatest and the smallest chosen number in an optimal solution. Consider two cases:

1) If there does not exist an unchosen number **c** in the sorted array from the interval <**a**, **b**>, a subsequence of consecutive elements is chosen and the proof is finished.

2) Else, determine which number out of **a** and **b** has the greater distance to the set of chosen numbers. We remove it, and pick **c** instead. The greatest absolute difference **M** is now decreased because **b** - **c** (or **c** – **a**) is less than **b** - **a**. By removing a number we have not increased the smallest absolute difference **m** because we have removed the greater of the two differences (distances). By adding **c**, the difference of its two chosen neighbours in a sorted array is divided in two parts, so **m** is not increased this way. We have thus proved that each optimal solution can be reduced to the consecutive subsequence in a sorted array.

Here is a simple solution with a time complexity of O($\mathbf{N}^2$) which sorts the array, for each of its substrings of the lenght **N** - **K** calculates the smallest and the greatest difference and returns the best solution. This solution is worth 50% of total points.

Notice that the greatest difference is actually the difference between the first and the last element of the substring and we can calculate it in a constant time complexity for each substring. If we keep the last **N** - **K** – **1** differences of consecutive array elements in a structure like balanced binary tree, the smallest difference can be obtained in constant time complexity, but the operations of inserting and deleting have a logarithmic time complexity. This way we lower the complexity to O(**N** log **N**). This solution is worth 70% of total points.

Notice that the elements of the array are limited to the interval of **R** integers. We can therefore use a counting sort in the complexity of O(**N** + **R**) to sort them. It is enough to calculate the frequency of each number in the interval and passing along the frequency array generate a sorted array.

Let us take a look at the data structure we use to keep the differences of consecutive elements of the array. If the newly inserted difference is smaller than another difference in the structure, the other one will never be the smallest becuase it will be deleted before the newly insterted one. Therefore, the differences can be kept in a monotonous deque with two ends. The smallest difference in the substring will be the one at the beginning of the deque. Since **N** - 1 differences of consecutive elements will be inserted and deleted exactly once, we lower the time complexity to O(**N** + **R**) and achieve full points.

Necessary skills: monotonous deque, counting sort

Category: data structures

COCI 2012/2013	Task DLAKAVAC
4th round, January 19th, 2012	**Author:** Anton Grbin

Let us present infected people in one day as an object which has a defined operator *. We want the operator to determine the infected people for the next day, based on two of the past days. Let us denote the first day with **B**. We denote the second and every **j**th day as **K(j).** Then we have:

K(1) = **B**,
K(2) = **B** * **K**(1) = **B** * **B**.

We can think of a day as an array of 0s and 1s such that there is 1 in the **i**th position if the **i**th person was infected that day. The operator can then be implemented in a complexity of O(**M** * **M**).

Let us take a look at the solution for day 3:

K(3) = **B** * **K**(2) = **B** * **B** * **K**(1) = **B** * **B** * **B** = $\mathbf{B}^3$.

Here we add an operator of exponentation as a shortened writing of consecutive multiplication. Let us take a look at the properties of this operator.

$B^i = B^{i-1} * B$

This is obviously right because we have defined the exponentation as a shortening for consecutive multiplication.

$\mathbf{B}^i * \mathbf{B}^i = \mathbf{B}^{i+i}$

This relation is also true for our operator *. It is now possible, using logarithmic exponentation, to complete the operation in the complexity of O(log **K** * **M** * **M**) where **K** is a day we are interested in.

The algorithm:

```
power(B, k) =
  if k is even,
    half = power(B, k / 2) if k != 0, or {0, 1, 0, 0, ..} if k = 0
    return half * half
  if k is odd,
    return power(B, k - 1) * B
```

If **k** becomes 0, we must return the object which will be the neutral element for the operator *. In our case it is {0, 1, 0, 0, ..}.

Necessary skills: arrays, mathematical analysis of the problem, logarithmic exponentiation

Category: math

COCI 2012/2013	**Task AKVARIJ**
4^{th} round, January 19^{th}, 2012	**Author:** Gustav Matula

The area below the aquarium bottom consists of **N** - 1 trapezoids. We will find the way to calculate the area of one trapezoid and then improve it using data structures for faster work with all of them. For the sake of simplicity, we calculate the area of the unflooded part below the water level **h**.

There are three cases for a trapezoid:

1) Water area is above the entire trapezoid.
2) Water area is below both upper vertices of the trapezoid.
3) Water area is between the upper vertices of the trapezoid.

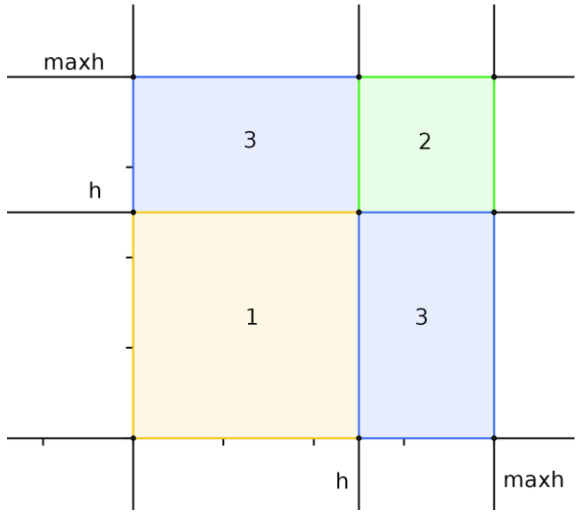
In the first case, the area is equal to the area of the entire trapezoid.

In the second case, the area is equal to the water height **h**.

In the third case we calculate the area in the form $A * \mathbf{h}^2 + B * \mathbf{h} + C$ (determining polynomial coefficients is left as an exercise to the reader).

In case there are more trapezoids, for a height **h** we can divide them into groups 1), 2), 3) above. From the first group we need the sum of all areas, from the second we need the number of trapezoids (the total area in this group is equal to **h** * number_of_trapezoids), and from the third we need the sum of corresponding polynomial coefficients, from which we can get a formula for the area of the group.

The problem is calculating the sum fast enough for each group. If for a given **h** we present each trapezoid as a point ($\mathbf{H_i}$, $\mathbf{H_{i+1}}$), groups become rectangles as shown below:



Queries on such 2D intervals can be done using a 2D Fenwick tree. In each element of the structure we store the sum of total areas, number of trapezoids and three sums of coefficients (A, B and C). This structure is simply maintained when we change the height, and the complexity of a query is O($\log^2$ **maxh**). The total complexity of the algorithm is O((**N** + **M**) $\log^2$ **maxh**).

There are alternative solutions. Instead of maintaining the structure of trapezoids we can maintain the array of solutions for each **h**. For each height, we are interested in the same three cases as before. If we have a trapezoid ($\mathbf{H_i}$, $\mathbf{H_{i+1}}$) (for the sake of simplicity assume $\mathbf{H_i} <= \mathbf{H_{i+1}}$), it contributes to the interval [0, $\mathbf{H_i}$] with its total area, to the interval ($\mathbf{H_i}$, $\mathbf{H_{i+1}}$) with its formula (see case 3), and to the interval [$\mathbf{H_{i+1}}$, **maxh**] with increasing the number of trapezoids by 1. The array of solutions can once again be achieved as a Fenwick tree or a tournament tree and the complexity of an operation is then O(log **maxh**), in total O(**N** log **maxh**).

Necessary skills: Fenwick tree or tournament tree

Category: geometry, data structures