CROATIAN OPEN COMPETITION IN INFORMATICS
2014/2015

2nd round, November 8th, 2014

solutions

Task MOBITEL	Authors: Antonio Jurić, Marin Tomić

Let us imagine for a moment that the grasshopper's mobile phone didn't fall into a puddle. In that case, the task comes down to a simulation of writing a word using the phone's keys. In order to achieve the aforementioned simulation, for each letter in the word, we add to the solution the number of pressed keys needed to get to that letter (according to the image from the task). Additionally, we have to be careful to separate pressing consecutive letters from the same key with '#'.

When we find a solution obtained by the above method, it is sufficient to replace every occurrence of key **x** with key **y** if key **y** acts like key **x**. This simple modification solves the grasshopper's problem.

For implementation details, consult the source code.

Necessary skills: loops, arrays, strings
Category: implementation, ad-hoc

Task UTRKA	Author: Ivan Paljak

A naive approach to solving this problem would be the following algorithm:

```
For each contestant from the list
     if the contestant's name is on the ranking list
          mark contestant
          cross out contestant from the ranking list

Output unmarked contestant
```

This implementation has a time complexity of `O(N^2*duljina_riječi)` and was sufficient for *50%* of total points.

To win all points, it was necessary to notice that the name of the contestant that didn't manage to finish the race is going to occur an **odd** number of times in the input, whereas the names of all the other contestants are going to occur an **even** number of times.

If we think this through, we will see that an analogous claim holds for every letter in the name of the wanted contestant. For example, if we observe the number of occurrences of initial letters of all the contestants, we will see that all the letters except the initial one of the wanted contestant occur an even number of times. This claim leads us to the following idea:

Let `cnt[index][letter]` denote the number of occurrences of the letter `letter` on position `index`. Then it is sufficient that, after we have filled out the matrix, find which letter occurs an odd number of times for each index. Those letter, starting with index 0, make up the final solution. Filling out this matrix of dimensions `27x21` can be done in `O(N*word_length)` which is fast enough for limitations from the task.

In the style of the main character of the sixth task -- "Can it be pimped up? Of course it can!"
Let us recall some properties of the binary operation `xor`. More specifically, the following properties are essential:

```
1) a xor b = b xor a
2) a xor a = 0
3) a xor 0 = a
```

Combining these properties, it is easy to deduce that it is sufficient, for every index, output `xor` of all letters from the input that occur in that index. We leave the formal proof for the reader to practise.

For the implementation of this solution, also of the complexity `O(N*word_length)`, you can consult the official solutions.

Necessary skills: loops, strings, xor
Category: ad-hoc

Task STUDENTSKO	**Author: Marin Tomić**

Let us examine a sample test (8 students, teams of 2 players):

```
3 2 1 6 7 5 8 4
```

Let us denote the students that must be in the same team with the equal numbers:

```
2 1 1 3 4 3 4 2
```

Ante can take any number from the sequence and relocate it. With these operations, in the minimal number of steps, he needs to obtain the sequence:

```
1 1 2 2 3 3 4 4
```

Let us find the longest subsequence of elements that are in a relatively good order in the sequence:

2 **1** **1** **3** **4** 3 **4** 2

This is actually the longest nondecreasing subsequence of the sequence. Now we can relocate all the elements that don't belong to this subsequence to corresponding locations, each in one minute.

It is easily shown that this solution is optimal and we leave this for the reader to practise. Finding the longest increasing subsequence is a typical problem that can be solved in either O($\mathbf{N}^2$) or O(**N** lg **N**) complexity using dynamic programming.

We recommend that you read the solution to this typical problem on this link, and for implementation details consult the source code.

Necessary skills: longest increasing subsequence
Category: greedy algorithms, sorting

Task BOB	**Author: Dominik Gleich**

The task is to find the number of rectangles with all of its squares equal. Let us try to first solve the simpler version of the task; there are only two types of squares: 0 and 1 and we need to count the number of rectangles that don't have a zero, in other words, consist of only 1's.

Let us try to count these rectangles in a way that we fixate their lower right corner. Furthermore, let us try to count from left to right, going down through rows. Additionally, let us define a[x][y] as the number of 1's until the first zero to the top of the matrix. (x is the vertical component, y horizontal)

Let us assume that we are calculating the solution for the lower right corner (x,y).
Let (x, z) be a pair so that it holds a[x][z] < a[x][y], that z < y and that z is maximal out of all possible z's. In other words, we want to know the first position to the left of our current position y so that the height of that column is smaller than the height of the column at our current position.

To continue, we need to notice that every rectangle with its right corner at current position (current y), and left corner at position z or to the left of it is actually a rectangle that has the right corner at position z, extended to column

y. Therefore, the number of rectangles that end at y and go over z are equal to the number of rectangles ending at z.

We still have to count the number of rectangles that have the left corner to the right of z. The number of such rectangles is (y - z)*a[x][y] because all columns up to z are higher than or equal to the column at a[x][y], which means that every rectangle height is possible, its width being from anything to y-z.

It is necessary to somehow maintain a structure that will give an answer to the question: first column on the left, smaller than the current column. This can be done using a stack or some of the more advanced data structures such as logarithmic or tournament tree.

Using the stack, we can do this in the following way (pseudocode):

```
while column on stack peak is larger than current column
    pop that column from stack
y <- column left on stack peak
push current column to stack
```

The complexity of this algorithm on stack is linear.
After we have solved this simpler version of the task, we still need to expand the solution for the original version of the task.
The expansion is fairly simple and is left for the reader to practise.

Necessary skills: stack
Category: ad-hoc

Task ŠUMA	**Author: Mislav Bradač**

Let us first solve a simpler version of this task where there are no two adjacent fields with trees of the same height and the same growth speed.

Let us examine two adjacent fields. The trees in those fields can be of equal height in at most one single moment. The moment when trees in two adjacent fields become of the same height will be called an event.

The number of events is smaller than **4N**. For each event, we can use DFS (or BFS) to spread from the field where trees have become of the same height and counting trees in a component, being careful that for a certain moment we do not iterate over one field more than once.

This algorithm has a complexity of O($\mathbf{N}^2$) because we will iterate over every field 4 times at most (once for each event which includes that field).

In the original task, a connected component of trees can grow together (if their initial heights and growth speed are equal). If we apply the previously described algorithm, we are left with a time complexity of O($\mathbf{N}^4$). This complexity is achieved in the example where there exists a large component growing together. This approach was sufficient for 30% of total points. The next idea that might come to one's mind is to compress the component of trees that grow together into one node and construct a graph where two nodes are connected if their corresponding components are adjacent in the initial matrix. This approach does not change the complexity because there can exist a component with many edges (order of magnitude $\mathbf{N}^2$), and during DFS for every event, we will be iterating over all of its edges. This approach, given some additional (non-deterministic) optimization, was sufficient for 50% of total points.

The solution for 100% of total points is based on the previous solution, but we will not be using BFS or DFS but union-find structure for event processing. Union-find structure will remember the parents of each node and the root of each connected set remembers the size of its corresponding set.

First we compress all connected trees that grow together into a single node, then calculate all events for each two adjacent nodes. The events are processed so that we connect the nodes in the union-find structure which in that moment become of equal height. During the event processing, we remember all the changes we made in the union-find structure in order to be able to restore the structure to the state before the event processing began. After the event processing, we restore the union-find structure to its initial state. This solution is of the complexity O($\mathbf{N}^2$ lg $\mathbf{N}$).

It is good to note that we don't need to use floating points numbers during calculations, sorting and comparison of events. Careless handling of data types with floating point resulted in wrong answers in some of the test cases. The official solution is an example of handling events without floating point numbers.

Necessary skills: union-find
Category: graphs

Task NORMA	**Author: Ivan Katanić**

The goal of this task is, for each two integers A and B (1 <= A <= B <= N), calculate the price of the subsequence [A,B] of input array X and to sum up the given values. Let us imagine that we are moving the right bound B from left to right and maintaining the array of solutions for current B, so that at position A of that array, call it array T, there is the price of subsequence [A,B].

A high-level algorithm would look like this:

```
initialize array T to 0
solution = 0
for B from 1 to N
     subsequences are expanded from [A,B-1] to [A,B] by adding
X[B]
     therefore refresh values in array T at positions 1 to B
     add to the solution T[1] + T[2] + … + T[B]
```

Let's imagine that every member T[A] of array T internally contains values m, M and L, minimal number of sequence [A,B], maximal number of sequence [A,B] and length of subsequence [A,B], respectively.

We need to have an efficient update of values (prices) in array T. Since the value of a member is the product of its internal values, let us observe how these are changing when incrementing B by 1 (moving to the right). Values L of each member with index smaller than or equal to B are incremented by 1.

Let P_m be the position of the rightmost member of array X that is to the left of B such that it holds $X[P_m] < X[B]$. Value m of all members of array T at position from the interval $[1,P_m]$ will be left unchanged while the value m of all members of array T on positions from the interval $[P_m+1,B]$ will become exactly X[B].
Similarly, P_M be the position of the rightmost member of array X that is to the left of B such that it holds $X[P_M] > X[B]$. Value M of all members of array T at position from the interval $[1,P_M]$ will be left unchanged while the value M of all members of array T on positions from the interval $[P_M+1,B]$ will become exactly X[B].

Therefore, it is necessary to implement a data structure that will allow the following operations:
increment_L(lo, hi, d_L) - increment value L by d_L to all members of the array at position from the interval [lo,hi].
set_m(lo, hi, new_m) - set value m to new_m to all member of the array at position from the from the interval [lo,hi].

`set_M(lo, hi, new_M)` - set value `M` to `new_M` to all member of the array at position from the from the interval `[lo,hi]`.
`sum_mML(lo, hi)` - return the sum of products of values `m`, `M` and `L` of all the members of the array at position from the interval `[lo, hi]`.

It turns out that the required operations can be efficiently achieved using a tournament tree. For this purpose, every node of the tree must contain the following values:

`len` - length of the interval of members of the array that the node is covering, for example, for leaves it holds `len=1`
`sm` - sum of values `m` of all members from the interval
`sM` - sum of values `M` of all members from the interval
`sL` - sum of values `L` of all members from the interval
`smM` - sum of values `m*M` of all members from the interval
`smL` - sum of values `m*L` of all members from the interval
`sML` - sum of values `M*L` of all members from the interval
`smML` - sum of values `m*M*L` of all members from the interval

How would incrementing values `L` to all members from the interval belonging to the node of the tree look like?
It is necessary to suitably change the value of each node in the following way:

```
sL = sL + d_L * len
smL = smL + sm * d_L
sML = sML + sM * d_L
smML = smML + smM * d_L.
```

Let’s observe how setting values `m` to all members from the interval belonging to the node of the tree looks like:

```
sm = new_m * len
smM = new_m * sM
smL = new_m * sL
smML = new_m * sML.
```

And, finally, setting values M to all members from the interval belonging to the node of the tree:

```
sM = new_M * len
smM = new_M * sm
smL = new_M * sL
smML = new_M * smL.
```

Because of the fact that all values in the nodes are sums, merging two nodes of a child for the purpose of calculating values of the parent node is simply the summation of the corresponding values from both nodes.

The nature of mentioned operations on the structure is such that it is necessary to use tournament tree with propagation. The details of this expansion are general and left out from this description, but can be looked up in the attached code.

We are still left with efficiently finding the rightmost smaller or larger number than the current `X[B]`. This can be simply implemented using a stack. Here we will describe finding the rightmost smaller value that is to the left of `X[B]`, and finding the rightmost larger value is done in a similar manner.

If we take into consideration that `B` is being moved from left to right, from `1` to `N`, then we are actually talking about the last member of array `X` that we passed, and is smaller than `X[B]`. For each `B` from stack we will pop all the numbers larger than or equal to `X[B]` since from now on they can't ever be someone's last smaller number (`X[B]` is smaller than them and is located after them). After that, stack's peak will be exactly member of array `X` that we were looking for, the last one smaller than `X[B]`. Before incrementing `B` and moving to the right, we push `X[B]` on stack. The given algorithm is of linear complexity because each member of the array is pushed and popped from stack at most once.

All operations on tournament tree are of the complexity O(log **N**), and they are done O(**N**) times, so the total complexity of this solution is O(**N** log **N**).

Necessary skills: tournament tree, propagation, stack
Category: data structures