CROATIAN OPEN COMPETITION IN INFORMATICS

2013/2014

ROUND 5

SOLUTIONS

COCI 2013/2014	Task LOZINKA
5th round, February 15th, 2014	**Author:** Anton Grbin

This is an implementation task. Firstly, we load the whole dictionary into the memory, then make a reversed copy of every word and go through the dictionary checking whether a reversed word exists. If the reverse word indeed exists in the dictionary, we must output its length and central character.

Notice that, because all the words are of odd length, the central character in the original and in the reversed word is going to be the same.

Necessary skills: strings, string reversal, string comparison
Category: ad-hoc

COCI 2013/2014	Task OBILAZAK
5th round, February 15th, 2014	**Author:** Marin Tomić

Firstly, we will enumerate the nodes in the tree in the following way: the root is number 1, its left child is number 2 and its right child is number 3. Generally, the left child of a node **t** is numbered with 2 * **t** and the right child is numbered with 2 * **t** + 1.

Mirko walks around visiting the buildings following this recursive algorithm:

```
visit(x)
if x < 2^K-1, visit(2 * x)
write down x on paper
if x < 2^K-1, visit(2 * x + 1)
```

We can construct a "reverse" recursive function which will reconstruct a tree for a given route of visits. We will use an array tree[] where the ith item is going to be a label of the ith node and a counter p which is initially set to 1.

```
return(x)
    if x < 2^K-1, return(2 * x)
    tree[x] = route[p]
    p = p + 1
    if x < 2^K-1, return(2 * x + 1)
```

In the end, we output items of the array tree[].

The task could have been solved without recursion, with noticing regularities in the route of visits of the tree. Consult the source code for this type of solution.

Necessary skills: recursion, for loop
Category: ad-hoc

COCI 2013/2014	**Task EKSPLOZIJE**
5th round, February 15th, 2014	**Author:** Antonio Jurić

A simple solution which literally simulates the described procedure of chain reaction (finds the first appearance of the explosion, deletes it, concatenates the rest of the string and repeats this procedure) is not fast enough because of the limitations (**1 <= |Mirko's string| <= 1 000 000**).

One of the faster solutions uses hashing. But, given the fact that the explosion consists of only **different** characters (uppercase and lowercase letters of the English alphabet and digits 0, 1, … 9), it is possible to come up with a solution in linear complexity which uses stack to keep track of certain data during the string analysis.

The string analysis is done in the following way: we scan Mirko's string character by character. Because the stack is used to quickly identify the explosion, it will keep track of the **position** in the explosion. During the analysis, **two** main situations can happen:

1. The current character is **equal** to a possible **beginning** of the explosion: mark this position and remember it (push it on the stack).
2. The currect character is **not equal** to a possible **beginning** of the explosion (and the stack is not empty):
    a. the current remembered position in the explosion on the stack **fits** the current character in the string: move this position forward in the explosion and push it back on the stack (it is possible that in this step we have reached the end of the explosion, which means we have located it in the string so it is necessary to mark the beginning and the end of the explosion for later deletion and then we don't push it back on the stack)

b. the current remembered position in the explosion on the stack **doesn't fit** the current character in the string: empty the whole stack (we leave this question to the reader, as to why we are discarding the whole stack in this situation)

When we finish scanning the string, it is necessary to delete the marked explosions in the string in one pass.
This way, using the stack, we have made sure that our algorithm works in situations when deleting an explosion and concatenating the rest of the string produces new explosions.

A special case is when the length of the explosion is 1. For implementation details, consult the source code.

Necessary skills: problem analysis, strings, stack
Category: ad hoc

COCI 2013/2014	**Task DOMINE**
5th round, February 15th, 2014	**Author:** Anton Grbin

This task is a classic example of dynamic programming.

Let the state be a function which maps some form of the board with potentially occupied fields in the last row and the number of dominoes into the best coverage we can possibly get. More specifically, we want to know the best coverage for every number of rows N, number of dominoes K and a set of occupied fields in the last row B.

The solution for a certain state can be calculated based on previous states so that we try to place dominoes in the last row in every 11 ways possible. Each of these ways is going to need some available fields in the previous row and will occupy some fields in the current row. The ways are represented in a bitmask where the first digit is the binary code of the place we're occupying in the previous row and the second digit is the binary code of the place we're occupying in the current row.

```
const mask_t cases[11] = {

  0x00,                                          // without domino placed
  0x11,  0x22,  0x44,  0x03,  0x06,     // if we place 1 domino
```

```
 0x17, 0x47, 0x33, 0x55, 0x66      // if we place 2 dominoes
};
```

Using these moves we can construct a relation. The number of states is 8 * N * K = 8 000 000, whereas the relation in our example is a constant, 11. The memory usage is exactly 8 * 1000 * 1000 long long types of data. This is 64MB.

You can find the source code for the solution which uses only 32 MB in the archive.

Necessary skills: bit masks, breaking a problem into pieces, clean implementation and code testing
Category: dynamic programming

COCI 2013/2014	**Task TROKUTI**
5th round, February 15th, 2014	**Authors:** Adrian Satja Kurdija, Marin Tomić

Considering three different lines, we can notice that they form a triangle if and only if their slopes are mutually different. The lines **i** and **j** have different slopes if $\mathbf{A_i}$ / $\mathbf{B_i}$ is different than $\mathbf{A_j}$ / $\mathbf{B_j}$ (be careful with division by zero). This leads us to our first solution with the complexity of O($\mathbf{N^3}$), where we test whether the slopes are different for every line triplet (i, j, k).

If we could find out how many lines there are with a slope different than the slope of a certain pair of lines, the task could be solved by fixating all possible pairs of lines (**i**, **j**) with different slopes and add the answer to the query for lines **i** and **j** to the solution.

Let equal(x) represent the number of lines with a slope equal to the slope of line x. Then the number of lines with a slope different than the slope of lines **i** and **j** is **N** - equal(**i**) - equal(**j**). The values of equal(x) can be preprocessed in the complexity of O(**N** lg **N**) with simple sorting, which brings us to the solution of complexity O($\mathbf{N^2}$).

For a solution valid for 100% of points, introduce two new functions: greater(x) and smaller(x) which tell us how many lines there are with slopes strictly greater or strictly smaller than the slope of line x. These functions can also be easily preprocessed in the complexity of O(**N** lg **N**) with the help

of sorting. The number of triangles where the line **i** belongs and has the middle slope size can be calculated as greater(**i**) * smaller(**i**). For the final solution, we only need to sum these values for each **i** from 1 to **N**. The total complexity is O(**N** lg **N**).

Necessary skills: preprocessing, sorting, basics of analytical geometry

Category: ad hoc

COCI 2013/2014	**Task LADICE**
5th round, February 15th, 2014	**Authors:** Luka Kalinovčić, Gustav Matula

The state of drawers and items can be represented by a directed graph where the nodes are drawers and the edges are items. If an item is located in drawer **A**, and its alternative is drawer **B**, the graph contains the edge **A** -> **B**. Given the fact that each drawer can contain a maximum of one item, the out-degree of all nodes is either 0 or 1, the path from any drawer is unambiguous and ends in an empty drawer or a cycle of drawers.

If drawer **B** is empty and there if an edge **A** -> **B**, it is possible to store the item from drawer **A** to drawer **B**, which corresponds to swapping edge **A** -> **B** with edge **B** -> **A**. In other words, reversing the associated edge.

If there is a path from a full drawer **A** to an empty drawer **B**, the drawer **A** can be emptied by repeatedly moving items, which corresponds to swapping edges on the path from **A** to **B**.

If a path from a drawer ends in a cycle, that drawer cannot be emptied.

We need a data structure which can tell us whether it is possible to empty a drawer (or if it's empty already) or, in other words, is there a path in the graph to an empty drawer.

Let us mark *final[l]* the empty drawer which is the end of the path from drawer *l*; additionally, if *l* is empty, then *final[l] = l*, and if such a path doesn't exist, then *final[l] = 0* (an empty drawer 0 is added as a mark for a cycle).

When adding a new item, the following scenarios are possible:

1. If the item is being added in the empty drawer **A** whose alternative is drawer **B**, then for each *l* where *final[l]* = **A**, we change into *final[l]* = *final[***B***]*.
2. If the drawer **A** needs to be emptied first and then add an item, the same thing happens.

A naive modification of the array final can be too slow for the limitations in this task.

The key thing is to notice how the drawers can be divided into classes considering the final drawer (drawers with equal final drawers are located in the same class), which enables us to implement the relation final efficiently with the union-find structure:
http://en.wikipedia.org/wiki/Disjoint-set_data_structure.
A cycle is formed if we add an item whose drawers are both located in the same class.

With the help of the structure, we use a function *find(l)* which corresponds to the aforementioned array *final[l]* and function *unite(a, b)* which performs the substitution described in 1.

When we have defined these functions, the following can be applied:

For an empty drawer, *find(l) == l*.
For a drawer that can be emptied, *find(l) != 0*.
A cycle is formed when adding items with drawers **A** and **B** if *find(***A***)* == *find(***B***)*, and if it is added in the structure as *unite(***A***, 0)*.
Otherwise a new cycle is not formed and we add the item as *unite(***A***,* **B***)* (if we are adding the item in the drawer **A**).
For implementation details, consult the source code.

Necessary skills: union-find, graph theory
Category: graphs