CROATIAN OPEN COMPETITION IN INFORMATICS 2012/2013

ROUND 3

SOLUTIONS

COCI 2012/2013	Task SAHOVNICA
3rd round, December 15th, 2012	**Author:** Adrian Satja Kurdija

There are multiple different approaches to this problem: we can use a matrix of characters, but we don't need to.

First method

Using two for loops, where one iterates over the total number of output rows, and the other (nested) over output columns, we output the appropriate characters. Here, we need a function to determine, given the current row and column (**r**, **c**), whether the character is red or white.

Notice that, if we number the rows and columns from 0 (in which case relations $0 \leq \mathbf{r} < \mathbf{R} * \mathbf{A}$ and $0 \leq \mathbf{c} < \mathbf{C} * \mathbf{B}$ hold), the transformation (**r**, **c**) → (**r** div **A**, **c** div **B**) results in the row and column of the corresponding chessboard cell (where $0 \leq \mathbf{r}$ div $\mathbf{A} < \mathbf{R}$, $0 \leq \mathbf{c}$ div $\mathbf{B} < \mathbf{C}$). A simple observation leads to the conclusion that the cell is red if the sum of the row and column (**r** div **A** + **c** div **B**) is even, and white otherwise.

Second method

We use a character matrix in which we draw one by one chessboard cell. We choose the current cell using two nested for loops and draw it, again using two nested for loops. Here we need to compute the starting coordinates of the current chessboard cell: they are (1 + **r** * **A**, 1 + **c** * **B**), where **r** and **c** ($0 \leq \mathbf{r} < \mathbf{R}$, $0 \leq \mathbf{c} < \mathbf{C}$) are the row and column of the cell, and the character matrix is indexed from (1, 1).

Necessary skills: nested for loops, simple mathematical observation

Category: ad-hoc

COCI 2012/2013	Task POREDAK
3rd round, December 15th, 2012	**Author:** Adrian Satja Kurdija

We need to read two arrays of **N** strings each and, for every two strings in the first array, find the positions of the same strings in the second array. If the positions are in the correct order, we add 1 to the result; finally, we output the result.

How can we find the position of a string in the second array? One method is using simple for loops, but the resulting time complexity is then O(**N**^{2}) for choosing all pairs of strings in the first array, times O(**N**) for finding the positions of the two selected strings in the second array (we will ignore the complexity of string comparison, which is proportional to string length), totalling O(**N**^{3}). For the given **N**, such a program would be too slow.

In order to reduce the complexity to O(**N**^{2}), after choosing a pair of strings, we need to immediately (in constant time) find the position of the **k**-th string from the first array in the second array. In order to do that, we need to find those positions in advance.

One method is to, before finding the solution, for each **k**-th string from the first array, use a for loop to find its position in the second array and store it in an auxiliary array. This sounds similar to the slow method above, but there is an important difference: we iterate over the second array O(**N**) instead of O(**N**^{2}) times. The total complexity is O(**N**) times O(**N**) for the described auxiliary array precomputation, plus O(**N**^{2}) for checking all pairs, totalling O(**N**^{2}), which is fast enough.

Even though it isn't necessary with the given constraints, readers are encouraged to devise an even faster, O(**N** log **N**) solution.

Necessary skills: string comparison, precomputing (computing auxiliary data before the main algorithm)

Category: ad-hoc

COCI 2012/2013	Task MALCOLM
3rd round, December 15th, 2012	**Author:** Adrian Satja Kurdija

A solution that, for each string, iterates over the previous **K** strings and counts the strings with the same number of letters, is too slow.

A faster solution reads a string, counts its letters – let us denote the number of letters with **L** – and then immediately, without another for loop, answers the question: how many strings, out of the previous **K**, have exactly **L** letters?

In order to find that number quickly, we need to keep an auxiliary array with the corresponding count for each **L**. This array must, of course, be maintained: upon reading a new string with length **L**, we increment the **L**-th element of the auxiliary array by one, and decrement the **L'-**th element by one, where **L'** is the length of the string "falling out" of the last **K** strings interval, i.e. not included in the set of friends of upcoming strings anymore.

This solution is actually based on the sweep-line principle: the imaginary scanner is scanning through the rankings list and processing events such as *new name added* and *name removed from friends set*.

Necessary skills: using auxiliary arrays to speed up algorithms

Category: sweep

COCI 2012/2013	Task AERODROM
3rd round, December 15th, 2012	**Author:** Adrian Satja Kurdija

It is possible to implement a solution that, for each person, computes (fast enough) the desk where the person will finish check-in soonest. It can be done using, for example, a *priority_queue* structure; details are left as an exercise to the reader. Such a solution has a complexity of O(**M** log **N**). Notice that this greedy solution is also optimal: if every person selects a desk that is optimal for them, it is possible to order all of them in such a way that they don't have to pointlessly wait for one another, leading to such behaviour being optimal for the whole team. Consider why that is correct.

In any case, the solution above isn't fast enough for **M** = 1 000 000 000. A solution that is fast enough, with complexity O(**N** log **M**), uses binary search: we need to find the earliest time in which the whole team can finish check-in, which requires being able to tell, for any time **T**, whether it is smaller or larger than the optimum – whether **M** people can finish check-in in time **T** or not.

How can we check that? For each desk **k**, we compute how many people that desk can process in time **T** (which is **T** div $\mathbf{T_k}$), and compute the sum of the obtained numbers. If the sum is greater than or equal to **M**, it is possible to process **M** people (under the assumption that they select desks using a sufficiently smart strategy); otherwise, it is obviously impossible.

The algorithm should be clear now: we keep an upper and lower binary search bound, select a number **T** which is the average of the two bounds, determine (as described above) whether it is larger or smaller than the optimal solution and, based on that, move the upper or lower bound to **T**, halving the interval of potential solutions until only one number remains.

Necessary skills: binary search

Category: binary search

COCI 2012/2013	Task HERKABE
3rd round, December 15th, 2012	**Author:** Adrian Satja Kurdija

First solution

From the given words we can build a trie, also known as a prefix tree (http://en.wikipedia.org/wiki/Trie). Imagine that we are in the root of the tree and can see **M** subtrees. In each subtree, the words begin with the same letter, and that letter is different for all **M** subtrees: we need to first choose all words from one of the subtrees, then all words from another subtree, and so on. Therefore, we can reduce the problem to **M** separate subproblems which can be solved recursively.

If we have determined, for the **k**-th subtree (using recursion), that words of that subtree can be ordered in $\mathbf{A_k}$ ways, then the total number of orderings of all subtrees is the product of those numbers ($\mathbf{A_1} * \mathbf{A_2} * \ldots * \mathbf{A_k} * \ldots * \mathbf{A_M}$). However, we also need to include the number of orderings of the subtrees themselves, which is **M!** and must be included in the product.

The prefix tree must be implemented carefully in order to be fast enough and not use up too much memory.

Second solution

Based on a similar idea, but simpler to implement (without prefix trees). We sort the words alphabetically. Next, we look at the first letter of all words and find **M** blocks such that all words in a block have the same first letter. As in the first solution, the result is **M!** times the product of solutions for individual blocks.

In the recursion, we need to find subblocks for each of the blocks. However, now we can ignore the first letter, since we know it is equal for all words in a block, so we can consider only the second letter. Analogously, in the deeper levels of recursion, we only need to consider the letters in positions corresponding to the current level. We conclude that the total complexity of the recursion is proportional to the total number of letters. Of course, the

recursion here will be parameterized by the lower and upper bound of the current block and the index of the letter we need to observe.

Necessary skills: recursion, tries

Category: strings

COCI 2012/2013	Task PROCESOR
3rd round, December 15th, 2012	**Author:** Ivan Katanić

Let us consider the **N** 32-bit registers as **N** * 32 binary variables. At first glance, the problem looks like a textbook 2SAT problem example (http://en.wikipedia.org/wiki/2-satisfiability). However, given the small time and memory limits, such a solution isn't efficient enough to obtain all points for the task.

Notice that, if any solution exists, at least one more solution must exist, and it can be obtained by inverting the bits of all the registers from the first solution.

It follows that the following algorithm is correct:

1. Find a still unset binary variable **b**.
2. Set the value of **b** arbitrarily (to either 0 or 1).
3. Set the value of all binary variables that were ever XOR-ed with **b**, since we can now determine their value unambiguously.
4. If there are no unset variables left, stop; otherwise, return to step 1.

If, in step 3, we try to set a variable that is already set to the opposite value, we have found a contradiction and there is no valid solution. Notice that setting another value in step 2 would again lead to a contradiction in the same step 3, since XOR implications are bidirectional.

In the beginning we can therefore set any variable to either 0 or 1, because there are at least two solutions with opposite values of all variables. After the other three steps of the algorithm, we have a smaller remaining set of unset variables which has no relation to the already set variables, so we can apply the same rule (there are at least two solutions) and set any variable to either 0 or 1 and repeat the procedure.

We will choose the unset variable and its value in step 1 in such a way to obtain the lexicographically smallest solution: we find the most significant unset bit of the first register which still has unset bits, set its value to 0 and continue with step 2.

The small memory limit requires implementing step 3 iteratively (using, for example, a queue). The time complexity is O(**N** + **E**).

Necessary skills: bitmask manipulation, queues

Category: ad-hoc