CROATIAN OPEN COMPETITION IN INFORMATICS 2012/2013

ROUND 2

SOLUTIONS

COCI 2012/2013	Task MORTADELA
2nd round, November 10th, 2012	**Author:** Nikola Dmitrović

Since mortadella prices are expressed in a nonstandardized format “**X dollars for Y grams**”, we should first convert them to a format that we can compare. One possible method is finding the price of a single gram of mortadella. It can be obtained by dividing the price with the mass. Finally, we have to find the minimum price of one gram and multiply it by 1000. The pseudocode is given below:

```
read(XNSC,YNSC);
read(n);

price:=XNSC/YNSC;

for i:=1 to n do
{
     read(Xi,Yi);

     if Xi/Yi<price then
          price:=Xi/Yi;
};

write(price*1000);
```

Necessary skills: input/output, for loop, finding minimum value

Category: ad-hoc

COCI 2012/2013	Task KRIZALJKA
2nd round, November 10th, 2012	**Author:** Adrian Satja Kurdija

We can use a triple-nested for loop to choose crossword rows in all possible ways. Then we arrange them into a 3 by 3 matrix. The last remaining step is checking whether the crossword columns match the remaining three words. The simplest method for this is finding the three strings corresponding to the matrix columns, sorting them lexicographically and comparing with the remaining three strings from the input (which are already sorted lexicographically).

There is another reason that makes the sorted order of the given words convenient: as soon as we find a matching crossword, we can immediately stop the search knowing that we have found the solution – provided that the outermost for loop chooses the first word index in order from 1 to 6, the middle one chooses the second word in the same order, and the innermost one chooses the third word, again in the same order, it is easy to see that the first matching crossword that we find will also be the first one lexicographically.

Necessary skills: nested for loops, array/matrix manipulation, sorting

Category: strings

COCI 2012/2013	**Task LANCI**
2nd round, November 10th, 2012	**Author:** Gustav Matula

Consider the situation where we have **L** chains and **K** open links. If **L** - 1 ≤ **K**, we can use those **K** links to connect all the chains together (any two chains can be connected using a single link, so we need **L** - 1 links to connect **L** chains).

If **L** - 1 > **K**, we do not have a sufficient number of open links, so we need to open more. If we open a link in the middle of a chain longer than three links, we will increase the number of chains by 1, which is obviously not optimal. Therefore, it is always best to remove links from either end of the chain. Furthermore, if a chain consists of single link, opening it reduces the number of chains by 1, which leads to a better solution. We conclude that it is best to take new links from either end of the currently shortest available chain, until we have enough open links (since the shortest chain will be the first to be completely taken apart and thus reduce the chain count).

Necessary skills: mathematical problem analysis, greedy algorithms

Category: greedy algorithms

COCI 2012/2013	Task POPUST
2nd round, November 10th, 2012	Author: Goran Žužić

What does an optimal solution for a given **K** look like? There are two possibilities:

1. **K-1** meals with the lowest **B** price and one meal with the lowest **A** price (out of the remaining meals)
2. **K** meals with the lowest **B** price, where one of them is chosen as first (subtracting its **B** price and replacing it with the **A** price – obviously, the one with the lowest $\mathbf{A}_i$ - $\mathbf{B}_i$)

Let us sort the meals by ascending **B** prices and solve the problem incrementally for ***K*** *= 1, 2, …,* ***N***. We will keep the current sum of **B** prices for the first **K-1** meals (sorted by **B**). Also, using a structure (such as a C++ STL set) we will keep the remaining, unselected, meals sorted by **A** (to be able to find the price of the first case), and another structure will keep the selected meals, sorted by $\mathbf{A}_i$ - $\mathbf{B}_i$ (to be able to find the price of the second case). The total complexity is O(**N** log **N**) if a structure query takes O(log **N**) time.

Necessary skills: application of data structures (STL or a similar library in another language, or a custom implementation)

Category: data structures, ad-hoc

COCI 2012/2013	**Task INFORMACIJE**
2nd round, November 10th, 2012	**Author:** Ivan Mandura

Let us build a bipartite graph with numbers on the left side and sequence positions on the right. An edge between nodes **(x, y)** exists by default, **except** if the provided descriptions make it **impossible** for number **x** to be in position **y**.

Each number can only appear in the intersection of all intervals describing that number, whether defining it as a local minimum or maximum.

Furthermore, we can determine for each position the range of numbers that can be placed in it. We will only consider intervals whose lower bound is less than or equal to the current position, and the right bound greater or equal. Possible numbers in that position range from the largest local minimum to the smallest local maximum of those intervals.

For an edge **(x, y)** to exist, the position **y** must be in the intersection of all intervals for **x**, and **x** must be in the allowed range for position **y**.

The only remaining problem is selecting a set of edges such that each number is assigned to exactly one position. It can be solved using a **flow/matching** algorithm. The problem has sufficiently small bounds to be solvable for all points even using the Ford-Fulkerson method.

Necessary skills: matching/flow algorithms

Category: graph theory

COCI 2012/2013	**Task INSPEKTOR**
2nd round, November 10th, 2012	**Author:** Ivan Katanić

Basic idea:
Let us divide the offices into **N**/**K** blocks such that each block contains **K** consecutive offices. If the companies inside a block are organized in a certain data structure (which we will describe later), moving a company in or out can be accomplished by changing a single block, that is, with complexity O(structure_delete + structure_insert). Mirko's stroll from **A** to **B** can be solved by manually checking the ending portions of the interval [**A**, **B**] which do not form a complete block (there will be at most 2*(**K**-1) such checks), and solving the remaining blocks, that are completely contained in [**A**, **B**] (there are at most **N**/**K** such blocks), by querying the corresponding structures. The total complexity of a stroll is then O(2*(**K**-1) + (**N**/**K**)*structure_query). In the complexity formulas above, 'structure_X' represents the complexity of operation X on the data structure.

The data structure:
Let us begin by calculating the account balance of company **i** on day **t**:

$$\mathbf{S}_{\mathbf{i},\mathbf{t}} = (\mathbf{t} - \mathbf{T}_\mathbf{i})*\mathbf{Z}_\mathbf{i} + \mathbf{S}_\mathbf{i} = \mathbf{t}*\mathbf{Z}_\mathbf{i} + (\mathbf{S}_\mathbf{i} - \mathbf{T}_\mathbf{i}*\mathbf{Z}_\mathbf{i})$$

Notice that the function mapping time to balance is linear, so our structure needs to support queries to find the line with the largest y value for a given x coordinate (in our case, **t**).
This can be done by constructing an upper convex hull of the lines and then using ternary search to find the maximum value for a given x.
However, since the x value (time) in consecutive queries is increasing, the maximum-valued line 'role' will also move from left to right, so it is sufficient to keep a pointer to the currently maximum-valued line and, given a new x, increment it while the next line to the right has a greater value for x.

In order to construct an upper convex hull out of the lines in time O(**K**), where **K** is the number of lines, we need to have the lines sorted by the coefficient multiplying x (the slope).
Therefore, our structure will keep track of:

1. the sorted array of lines,
2. the upper convex hull formed by those lines,
3. the pointer to the currently maximum-valued line.

Deleting and inserting a line is done by traversing the sorted array, adding the new line and deleting the old one, which can both be done in O(**K**). After that, we reconstruct the convex hull and reset the pointer to the first line in the hull.

The complexity of a single query cannot be predicted since it depends on the position of the pointer on the hull, but we can be certain that the pointer will be incremented at most **K** times (until it reaches the last line) before the next convex hull rebuild, which resets it. Therefore, if $\mathbf{P_i}$ is the number of pointer resets (i.e. insert and delete queries), the total complexity of all queries on block **i** is O(**K***($\mathbf{P_i}$ + 1)) = O(**K***$\mathbf{P_i}$).

Total complexity:
Each operation of moving a company in (and moving the existing company out) requires changing a single block with complexity O(**K**), while the total complexity of all queries is the sum of query complexities for all blocks: O(**K***$\mathbf{P_1}$ + **K***$\mathbf{P_2}$ + … + **K***$\mathbf{P_{N/K}}$). Since the sum of $\mathbf{P_i}$ is at most **M**, the complexity equals O(**M***__K__). The total complexity is then O(**M***__K__ + **M***__N__/**K** + **M***__K__); if we select **K** = sqrt(**N**), this equals O(**M***sqrt(**N**)).

Necessary skills: block manipulation, linear geometry

Category: data structures, geometry