CROATIAN OPEN COMPETITION IN INFORMATICS

3rd ROUND

SOLUTIONS

COCI 2010/11	**Task TABLICA**
3rd round, December 11th, 2010	**Author:** Tomislav Grbin, Frane Kurtović

There are only four different rotations of the given matrix so computing the optimal solution is straightforward - it is necessary to find the maximum of values for all rotations and keep track of the ordinal number of the rotation so we can output it. Keep in mind that if there are several rotations which give the same maximum solution, the smallest number should be output.

If this is the matrix given in the problem statement:

$$\begin{array}{|c|c|}\hline \mathbf{A} & \mathbf{B} \\ \hline \mathbf{C} & \mathbf{D} \\ \hline \end{array}$$

By rotating once, twice, and three times we get:

$$\begin{array}{|c|c|}\hline \mathbf{C} & \mathbf{A} \\ \hline \mathbf{D} & \mathbf{B} \\ \hline \end{array} \Rightarrow \begin{array}{|c|c|}\hline \mathbf{D} & \mathbf{C} \\ \hline \mathbf{B} & \mathbf{A} \\ \hline \end{array} \Rightarrow \begin{array}{|c|c|}\hline \mathbf{B} & \mathbf{D} \\ \hline \mathbf{A} & \mathbf{C} \\ \hline \end{array}$$

These four matrices generate the following values: $\frac{A}{C} + \frac{B}{D}$, $\frac{C}{D} + \frac{A}{B}$, $\frac{D}{B} + \frac{C}{A}$, $\frac{B}{A} + \frac{D}{C}$.

One trick is to notice is that by multiplying each of these numbers by **A* B * C * D** we simplify double comparisons to integer comparisons.

Necessary skills:

Basic algebra

Tags:

Ad-hoc

COCI 2010/11	Task ZBROJ
3rd round, December 11th, 2010	Author: Goran Gašić

The key observation is that Perica can get the minimum number by replacing all occurences of digit six with digit five. Likewise, he can obtain the maximum number by replacing all occurences of digit five with digit six.

The simplest way to do this is to read in the input as strings and simply carry out the digit substitutions before converting them to integers and calculating the results.

An alternative is to read the numbers as integers and do direct integer manipulation to extract digits and make necessary substitutions.

Necessary skills:

Strings, basic algebra

Tags:

Ad-hoc

COCI 2010/11	**Task EKIPA**
3rd round, December 11th, 2010	**Author:** Matija Osrečki

In order to find a simple and elegant solution, it is important to notice that since each student can participate in a single category, but multiple students can compete in the same category, and each student has applied for all categories, it suffices to determine for each student the category which they know best. Then we can select the best **K** students, judging only by the category which they are best in.

Why is that solution correct? Precisely because any number of students can compete in the same category. If a student is selected to participate in a category, other students are not restricted in their category choice - only the selected student is prevented from competing in any additional categories. It follows that we have to determine for each student the category which they know best.

There are two possible implementations; one requires sorting, while the other one doesn't.

Solution:

For each student we determine the category which they know best. Now we have each student's knowledge represented by a single number. We can simply sort students by that number and select the first **K** students from the sortes dequence.

Time complexity is O(**N** * **M** + **N** * log **N**). Memory complexity is O(**N**).

Alternative solution:

We can take advantage of the fact that students are already sorted by categories. For each category we keep a pointer to the first (best) student. In each of **K** steps we iterate over all categories and determine the category that contains the pointed-to student with most knowledge. We add that student's knowledge to the result and move the pointer to the next best student in that category (the following one, since they are already sorted by category). It is also necessary to keep track of students that have already been selected, since they must be skipped if encountered again.

Time complexity is O(**N** * **M**). Memory complexity is O(**N** * **M**).

Necessary skills:

Greedy algorithms, sorting (not necessary, but simplifies the solution)

Tags:

Greedy algorithms

It is important to notice that if it is possible to delete **K** rows from the top of the matrix leaving no two equal columns, the condition will also be satisfied if only **K**-1 rows are deleted. Hence the maximum **K** can be found using binary search.

Now we need an efficient method to check, for a given **K**, whether it is possible to delete first **K** rows leaving no two equal columns. If we read the columns as strings from the bottom up, a good idea is to sort them lexicographically. That way, any two equal columns will be next to each other, so we only need to check adjacent columns for equality up to length **K**.

The described solution can be implemented with complexity O($\mathbf{N}^2$ * log **N**), where **N** = max(**R**, **C**).

An alternative solution could use hashing of individual columns, reducing column comparison to simple integer comparison (reducing complexity in the process). The details of this solution are left as an exercise for the reader.

Necessary skills:

Binary search, string comparison, sorting, hashing

Tags:

Binary search, string manipulation

The key observation follows: instead of calculating the sum of difference values of all contiguous subsequences, we can separately calculate the sum of all maximum values and all minimum values of the subsequences, subtracting the second from the first number to obtain the solution. It is not difficult to show that this approach is correct, since the difference value of a given subsequence is defined as the difference between its maximum and minimum value, thus the observation follows from addition associativity and commutativity. Here we will only describe a method to find the sum of maximum values; the sum of minimum values is obtained with a completely analogous algorithm.

Our approach is as follows: for each element of the provided sequence we shall compute the number of contiguous subsequences that contain that element as their maximum. In order to resolve ambiguity, we shall define the first out of multiple equal maximal elements as the maximum of a subsequence.

If the position of the current element is **K**, and its value equals **X**, we need to find the first element to the left of it with value greater than or equal to **X** and the first element to the right with value strictly greater than **X**. If the two elements satisfying the given conditions are in positions **A** and **B** respectively, then **X** is the maximum element in exactly those contiguous subsequences that start at positions between **A**+1 and **K** (inclusive) and end at positions between **K** and **B**-1 (inclusive). There are exactly (**K**-**A**)*(**B**-**K**) such subsequences, since we multiply the number of possible choices of the first element (**K**-**A**) with the number of possible choices of the last one (**B**-**K**) in the subsequence. Hence the sum of maximum values is increased by **X***(**K**-**A**)*(**B**-**K**).

Now we only need an algorithm to find the first element to the left of the current one that is greater than or equal to it (the algorithms to find the first one tho the right that is greater, as well as the other combinations for finding minimum values, are completely analogous). We shall traverse the sequence from left to right, keeping a stack of pairs (value, position) containing some previous elements of the sequence. The stack will be kept in decreasing order of the values. Processing of an element with value **X** consists of popping all elements with values strictly less than **X** from the stack. The element that remains at the top of the stack (if any) is precisely the first one to the left of **X** that is greater than or equal to it. After that the element **X** is pushed on top of the stack and the algorithm continues with the next element of the sequence.

Necessary skills:

Monotone stack

Tags:

Data structures

COCI 2010/11	**Task MONO**
3rd round, December 11th, 2010	**Author:** Luka Kalinovčić

Let us denote by **M** the rectangle with minimal area that completely contains the provided polygon. We intersect **M** with the least possible number of vertical lines such that one of them passes through each polygon vertex. The lines divide the polygon into rectangles, so its area can be computed as the sum of areas of individual rectangles.

Note that the area of a single rectangle can be computed using the inclusion - exclusion principle. The area equals the difference of areas above the lower and upper edge.

Furthermore, the sum of their areas can be obtained by traversing the polygon anticlockwise. Whenever we traverse a horizontal edge, we add the area above it to the sum, with the sign depending on the direction of traversal. Since the uppermost edge will be traversed from right to left, we will add areas in that direction with the negative sign.

Let us select a cell from the table as a potential upper left corner of the rectangle **M**. Next we determine a letter on one of the cells contained in the polygon specified by **M**. The polygon is monoliteral only if all its cells contain that same letter, i.e. if the number of incidences of that letter is equal to its area.

The number of incidences of a letter within a polygon can be calculated in an analogous way to calculating its area. The only difference is that instead of adding areas, we add the number of incidences of the required letter. A matrix can be precomputed containing data enabling us to answer such queries in constant time.

Finally, the total number of monoliteral polygons is obtained by running the described algorithm for each potential upper left cell in the table. The time complexity is O(**R** * **C** * **V**).

Necessary skills:

Inclusion - exclusion principle

Tags:

Computational geometry