

CROATIAN OPEN COMPETITION IN INFORMATICS

2nd ROUND

SOLUTIONS

| | |
|---|--------------------------------|
| COCI 2010/11 | Task PUŽ |
| 2nd round, November 13th, 2010 | Author: Stjepan Glavina |

Calculating the snail's position day by day would be too slow, since the expected result can be very large.

During one day and one night, the snail climbs exactly **A - B** meters.

His entire trip consists of **X** days and **X** nights, plus the last day in which he reaches the top of the pole.

Therefore, our solution is the smallest integer **X** for which $X * (A - B) + A \geq V$ holds.

Solution:

It's now easy to find a direct expression for **X**:

$$X * (A - B) + A \geq V$$

$$X * (A - B) \geq V - A$$

$$X = \lceil (V - A) / (A - B) \rceil$$

$$X = \lceil (V - A + A - B - 1) / (A - B) \rceil$$

$$X = \lceil (V - B - 1) / (A - B) \rceil$$

We use integer division, and print **X+1** as a result.

Alternative solution:

We can use binary search to solve the problem. We start with some potential solution **X'**, and check if the inequality holds. If the inequality doesn't hold, we must use a smaller **X'**. If it holds, we can increase **X'** and try again.

This gives us an $O(\log V)$ solution.

Necessary skills:

Inequality solving, binary search

Tags:

Mathematics, binary search

| | |
|---|-------------------------------|
| COCI 2010/11 | Task NAPOR |
| 2nd round, November 13th, 2010 | Author: Matija Osrečki |

This problem can be divided into two subproblems: extracting the numbers, and sorting them.

Extracting the numbers:

First thing to notice is that numbers that appear can have up to 100 digits, and cannot fit within a 64-bit integer data type (*long long* in C++, *int64* in Pascal). We must have our own representation of a number that big, and the simplest way to do this is to store them as strings.

We process text line by line, and traverse each line from left to right, keeping track of consecutive sequences of digits. We are going to need the following variables:

- **num** - array of found numbers
- **X** - part of the number found so far
- flag **S**

We use flag **S** to handle leading zeros. If **S**=0, we haven't found any digits so far. If **S**=1, we have found a sequence of zeros. While **S**=1, we don't add digits to our number **X**. We start doing that only after having found some non-zero digit, and mark this by setting **S**=2. We have to be careful to add numbers that are equal to zero. The exact algorithm is shown below.

For each character **c** in left to right order:

- if **c** is a letter
 - if **S** = 0 - do nothing
 - if **S** = 1
 - add zero to **num**
 - **S** = 0
 - if **S** = 2
 - add **X** to **num**
 - **X** = empty string
 - **S** = 0
- if **c** is a digit
 - if **S** = 0 and **c** = '0'
 - **S** = 1
 - **c** <> '0' or **S** = 2

■append **c** to **X**

■**S** = 2

Notice that the algorithm above won't add to array **num** any number located at the end of the line. We must check for this ourselves, or append any letter to the line before running the algorithm.

Sorting the numbers:

In order to sort the array **num**, we need a way of comparing two numbers stored as strings. If strings are not of equal length, it's easy to see that the shorter one is smaller. Otherwise, we need to find the leftmost position in which they differ, and compare characters at that position. Of course, if they are equal, we won't find such a position.

Once we have a compare function for our numbers, we must use some sorting algorithm. Since there are at most 500 numbers in input text, any algorithm will be fast enough. We can use a simple one, like bubble sort or insertion sort.

Necessary skills:

Parsing, big number comparison, sorting

Tags:

Parsing, sorting

| | |
|---|----------------------------|
| COCI 2010/11 | Task IGRA |
| 2nd round, November 13th, 2010 | Author: Goran Gašić |

Observe that, if Slavko loses the game with the most beautiful word, he could not have won. Therefore, it is sufficient to come up with a strategy which ensures that he constructs the most beautiful word at the end.

To accomplish the above strategy, Slavko is required to take **some** lexicographically smallest letter from the remaining sequence on each turn. Otherwise, if some other letter is to be taken, it is obvious that the final word cannot be lexicographically smaller than the one obtained by applying the above mentioned strategy.

In case of a tie, the **rightmost** such letter should be taken. The proof follows.

The proof:

Assume that Slavko takes the next move. Let **the chosen letter** denote the rightmost lexicographically smallest in the remaining sequence, and let **K** be the number of remaining letters to the right. There are three cases:

1. The number of remaining lexicographically smallest letters in the sequence is less than or equal to **K**.

Mirko has **K** letters to take before reaching **the chosen letter**. In the meantime, Slavko can take **all** lexicographically smallest letters (since none of these appear to the right of the chosen one), in arbitrary order.

2. The number of remaining lexicographically smallest letters in the sequence is greater than **K** and Mirko takes **the chosen letter** on his next turn.

If Slavko **can take all** lexicographically smallest letters from the sequence, he must, obviously, take **the chosen one** as well (otherwise Mirko would take it).

If Slavko **cannot take all** lexicographically smallest letters from the sequence, it is irrelevant which one of them he takes next (because sooner or later he will be forced to let Mirko take some lexicographically smallest letter, but it need not be now). Thus, he can take **the chosen letter**.

3. The number of remaining lexicographically smallest letters in the sequence is greater than **K** and Mirko does not take **the chosen letter** on his next turn.

Assume that Slavko does not take **the chosen letter** on his current turn, instead taking another one. Nevertheless, using the strategy above, he will need to take it before Mirko does. Therefore, Slavko can swap the two moves, taking **the chosen letter** on this turn and taking the other one (which will still be available) later.

Necessary skills:

Proof of correctness of a greedy algorithm

Tags:

Greedy algorithms

| | |
|---|-------------------------------------|
| COCI 2010/11 | Task KNJIGE |
| 2nd round, November 13th, 2010 | Author: Adrian Satja Kurdija |

Observe that the optimal rearranging protocol will be as follows: choose a book numbered **K**, then put books **K**, **K-1**, **K-2**, ..., 2, 1 (**K** through 1) to the top of the stack. Following is the proof of optimality for such a protocol.

If we put book numbered **K** on the top first, then it is certain that we will have to, at a later point, put each book with number less than **K** on the top. Otherwise, that book would end up below book **K** in the final array.

Similarly, if we previously put book **K** on top during our optimal reordering protocol, no other book with a number greater than **K** is worth putting on top, because we would have to put book **K** on top again (otherwise, book **K** would appear below this book in the final array). However, there is no point in moving book **K** on top twice, since the first movement could be ignored to get a shorter protocol, which means the former one was not optimal, contrary to the initial assumption. It follows that we should not put any book greater than **K** on top.

Therefore, if **K** is the number of the book we put on top at the start of an optimal protocol, we must put all books numbered less than **K**, and no other book. It is now clear that these books must be put in order **K** through 1.

After these movements, the first **K** numbers will be 1 through **K**, so we are interested if the remainder of the array is sorted making numbers **K+1** through **N** follow. In order for this to be true, all numbers greater than **K** must form an increasing subsequence in the original array because, by moving numbers 1, 2, ..., **K**, the relative ordering of other numbers is preserved, and this should be increasing in the final array.

Thus, we can let **K** be any number such that **K+1**, **K+2**, ..., **N** form an increasing subsequence in the initial array. Clearly, since we seek a solution with a minimal number of movements, we will choose the smallest **K** such that the above conditions hold.

Note that if **K** has the above property, then **K+1** has this property as well, because if **K+1**, **K+2**, ..., **N** form an increasing subsequence, then this is also true for **K+2**, **K+3**, ..., **N**. Thus, we can find the minimal **K** using binary search, or we can look at

numbers **N**, **N-1**, **N-2**, ... and find the greatest of them which does not form a decreasing subsequence with the previous elements of the array.

Necessary skills:

Pattern observation, proof of correctness

Tags:

Ad-hoc, binary search

| | |
|---|--------------------------------|
| COCI 2010/11 | Task LUNAPARK |
| 2nd round, November 13th, 2010 | Author: Stjepan Glavina |

If the table has an odd number of rows, it is possible to route the coaster right to the end of the first row, then one cell down, left to the beginning of the second row, and one cell down again. By doing this, we can visit all cells in the first two rows and reduce the problem to a table without them. The reduced table again has an odd number of rows, so we can repeat the same procedure until we are left with only one row. Then we can simply move right to the end of the last row, in the process visiting all remaining cells.

Thus we have shown that with an odd number of rows it is possible to visit all cells, making that the optimal solution. A similar solution is possible in case we have an odd number of columns.

The most difficult case is a table with an even number of both rows and columns. We shall prove that it is impossible to visit all cells in that case.

Let us visualize the table as if the cells were coloured black or white, in a chessboard pattern. The upper left and lower right corners are coloured white. Every path between those two corners first enters a black cell, then white, black again, and so on until ending in a white cell. Thus, it enters an equal number of white and black cells. Since the path starts at the top left corner, which is white, in the beginning there are fewer unvisited white cells than there are black cells. It follows that not all cells can be visited.

If we leave a single black cell unvisited, we will be able to visit all remaining cells. On the other hand, if we leave a white cell unvisited, we will also be unable to visit two black cells, which is obviously not optimal.

We can select any black cell to leave unvisited. It is optimal to choose the one with the smallest amusement value. All that remains is finding a route that will visit all cells except for the chosen one.

Solution:

We can visualize our algorithm as if two tokens were placed on the table, one in the upper left and another one in the lower right corner. We alternate moving one and the other until they meet in a cell. The two tokens' paths together form the solution.

If the chosen cell isn't in one of the first two rows, we can move the first token right to the end of the first row, then one cell down, left to the beginning of the second row, and one cell down again. Now we can ignore those two rows and continue.

Likewise, if the chosen cell isn't in the last two rows, we can backtrack using the lower token - left to the beginning of the last row, then one cell up, right to the end of the second-to-last row, and one cell up again. Now we can ignore the last two rows.

An analogous process can be used to visit the first or last two columns if they don't contain the chosen cell.

By repeating this procedure we will eventually reduce the table to only two rows and two columns. It is trivial to solve this case - we select the only possible route for the first token (right-down or down-right, the one not containing the chosen cell). Now the tokens have finally met in a single cell.

This can be implemented by keeping two direction arrays, one for each token. When the tokens meet, the first tokens' array contains the first part of the solution. We only need to reverse the order and direction of the second tokens' directions to obtain the rest.

Alternative solution:

If the table contains only two rows, it is simple to find a solution. Otherwise, if the chosen cell isn't in the first two rows, they can be eliminated as described above.

On the other hand, if the chosen cell is in one of the first two rows, we can visit all cells of the first three rows except for the chosen one. Now we can ignore the first three rows, so the table is reduced to one with an odd number of rows, the solution of which is described above.

The details of this solution are left as an exercise for the reader.

Necessary skills:

Proof of correctness of a greedy algorithm, pattern observation, reduction of a problem to a smaller one

Tags:

Ad-hoc, greedy algorithms

| | |
|---|--------------------------------|
| COCI 2010/11 | Task CRNI |
| 2nd round, November 13th, 2010 | Author: Stjepan Glavina |

Let us first count, for each cell, how many black rectangles contain that cell as their lower right corner.

Select a row **R**. We shall define the height of a column as the number of consecutive black cells in that column, counting up starting from row **R**. These column heights form a histogram. We traverse the row **R** from the first to the last column, while keeping track of "rising stairs" of black columns from the first to the current column. The number of cells contained in the "stairs" minus one equals the number of black rectangles that contain the current cell as their lower right corner. Minus one comes from the fact that we don't recognize a 1 by 1 rectangle (a single cell) as a black rectangle.

We can use a similar approach to count black rectangles that contain each cell as their lower left, upper right, or upper left corner. Using this data it is easy to calculate the number of rectangles contained in each quadrant of the table for each cell as the origin.

For any two non-overlapping black rectangles, we can draw a vertical line, horizontal line, or both a vertical and horizontal line between them.

The number of rectangle pairs separated by a vertical line (case **A**) can be calculated by selecting a column to contain the right edge of the left rectangle. Then the other rectangle will be to the right. Those two values can be calculated from previously computed values.

We can use an analogous strategy to find the number of pairs separated by a horizontal line (case **B**).

It is also possible to calculate the number of rectangle pairs separated by both a vertical and horizontal line (case **C**). For example, if we select a cell as the lower right corner of a rectangle, its pair will be in the lower right quadrant from that cell. The final solution is $\mathbf{A} + \mathbf{B} - \mathbf{C}$, because case **C** is counted twice in cases **A** and **B**. The whole computation can be carried out with complexity $O(\mathbf{N}^2)$.

Necessary skills: Dynamic programming, keeping a histogram and its area in linear time

Tags: Dynamic programming, counting, inclusion-exclusion principle