CROATIAN OPEN COMPETITION IN INFORMATICS

2013/2014

ROUND 3

SOLUTIONS

COCI 2013/2014	Task RIJEČI
3rd round, December 7th, 2013	**Author:** Marin Tomić

If we tried to simulate what Mirko is doing, we would have exceeded both the time and memory limit because the words that Mirko reads out on the screen can consist of hundreds of thousands (or even millions) of characters, even for small **K**. That's why we won't simulate Mirko's process, but count the number of letters A and B in the following way:

- initially, number_of_A = 1 and number_of_B = 0
- in one step we get one letter A for each letter B we've had so far, and one letter B for each letter A and each letter B we've had so far; therefore: new-A = number_of_B and new_B = number_of_A + number_of_B

Necessary skills: for loop, calculation

Category: for loop

COCI 2013/2014	Task OKVIR
3rd round, December 7th, 2013	**Author:** Adrian Satja Kurdija

The easiest way is to first fill out the whole chessboard with characters # and ., the dimensions being (**U + M + D**) x (**L + N + R**), and then typing in Mirko's crossword puzzle. For both operations, we need a double for loop iterating through the matrix (or submatrix).

As for filling out the chessboard, the field (**i**, **j**) will be # depending on the parity of **i + j**. As for typing in Mirko's crossword puzzle, the field (**i**, **j**) of Mirko's crossword puzzle from the input will be typed into the (**U** + **i**, **L** + **j**) field of our chessboard.

Necessary skills: matrix operations

Category: ad hoc

COCI 2013/2014	Task REČENICE
3rd round, December 7th, 2013	**Author:** Marin Tomić

Since there will always be an existing solution less than 1000, we can simply try and put each number from the interval [1, 999] in the sentence and check

whether it is the possible solution. For a sentence with the number **x** in it to be valid, the following condition must be met:
length of the name of number **x** + sum of word length in the sentence = **x**

Now the only problem is to create a function which will determine the names of numbers. We can do this in the following way:

- create three arrays, **special[]**, **tens[]**, **hundreds[]**
- the array **special** is going to consist of names of numbers 1 to 19, i.e. **special[11]** is going to be "eleven"
- the array **tens** is going to consist of names of tens, i.e. **tens[7]** is going to be "seventy"
- identically, the array **hundreds** is going to consist of names of hundreds, i.e. **hundreds[3]** is going to be "threehundred"
- now we implement naming the numbers as described in the task

Pseudocode:
```
name(x)
    answer = ''
    if the hundreds' digit of x is not 0:
        answer = answer + hundreds[hundreds' digit of x]
    remove the hundreds' digit from x
    if x is from the interval [1, 19]:
        answer = answer + special[x]
    else:
        if the tens' digit of x is not 0:
            answer = answer + tens[tens' digit of x]
        if the single digit of x is not 0:
            answer = answer + single[single digit of x]
```

Consult the solution code for further implementation details.

Necessary skills: string operations

Category: brute-force algorithms, strings

COCI 2013/2014	**Task KOLINJE**
3rd round, December 7th, 2013	**Author:** Adrian Satja Kurdija

People from 1 to **N** are going to get, respectively, B_1X, B_2X, …, B_NX kilos of ham, given an **X**. We need to find **X** such that the sequence is correct; from it, it is easy to calculate the total number of distributed kilos.

We want the following to be true: $A_1 + B_1X \geq A_2 + B_2X \geq \ldots \geq A_N + B_NX$.

Let us analyze the first inequation; the rest are analyzed analogously. From A_1 +

$\mathbf{B_1X} \geq \mathbf{A_2} + \mathbf{B_2X}$ follows
$(\mathbf{B_1} - \mathbf{B_2})\mathbf{X} \geq \mathbf{A_2} - \mathbf{A_1}$.

We'd like to divide this inequality by $\mathbf{B_1} - \mathbf{B_2}$ so only **X** is left on the left side. However, we need to be careful!

If $\mathbf{B_1} - \mathbf{B_2} = 0$, we cannot make the division, but the inequality turns into $0 \geq \mathbf{A_2} - \mathbf{A_1}$. In this case, if $\mathbf{A_2} - \mathbf{A_1} > 0$, we output -1 because this is a contradiction. Otherwise, the inequality is valid so we move onto the next one.

If $\mathbf{B_1} - \mathbf{B_2} \neq 0$, we can make the division, but dividing by a negative number changes the inequality signs so we either get $\mathbf{X} \geq (\mathbf{A_2} - \mathbf{A_1}) / (\mathbf{B_1} - \mathbf{B_2})$ or $\mathbf{X} \leq (\mathbf{A_2} - \mathbf{A_1}) / (\mathbf{B_1} - \mathbf{B_2})$. In one case we got the lower, and in the other the upper bound for **X**.

Repeating this operation for all **N** – 1 inequalities, we gathered some lower and upper bounds for **X**. Of all the lower bounds, we are interested in only the greatest (because it implies all the rest), and of all the upper bounds, we are interested in only the smallest (because it implies all the rest). If between those two bounds there is a number, that is, if the lower bound is smaller or equal to the upper, a solution exists: we can use the average of the lower and upper bound as **X**. Otherwise, the solution does not exist.

The task is also solvable with binary search; we leave the details for the reader to practise.

Necessary skills: mathematical problem analysis, mathematical inequality basics

Category: ad hoc

COCI 2013/2014	**Task PAROVI**
3rd round, December 7th, 2013	**Author:** Marin Tomić

Since this is all about big numbers, obviously calculating the distance for each pair individually is not fast enough. Nevertheless, we'll do it in a very specific way.

We won't go through all numbers with two nested loops, but create a function f(prefix1, prefix2, sum) which will build the numbers digit by digit.

For example, a pair of numbers (32, 1689) would be built by adding digits, respectively, (0, 1), (0, 6), (3, 8), (2, 9).

Function f would look like this:

```
f(prefix1, prefix2, sum):
    if prefixes are completely built, return sum

    let (x, y) go through all possible pairs of digits:
        if you can add x on prefix1 and y on prefix2:
            call f(prefix1 * 10 + x, prefix2 * 10 + y, sum + |x − y|)
```

This solution is not quicker than the two nested loops solution, but it can be tweaked. First we will get rid of the sum parametar from the function.

We will build a function num(prefix1, prefix2) which, for given prefixes of the first and second number, calculates the number of ways to "finish" these two numbers.

```
num(prefix1, prefix2):
    if prefixes are completely built, return 1

    sol = 0
    let (x, y) go through all possible pairs of digits:
        if you can add x on prefix1 and y on prefix2:
            sol = sol + num(prefix1 * 10 + x, prefix2 * 10 + y)
    return sol
```

Now the function f can be transformed as follows:

```
f(prefix1, prefix2):
    if prefixes are completely built, return 0

    sol = 0
    let (x, y) g through all possible pairs of digits:
        if you can add x on prefix1 and y on prefix2:
            sol = sol + num(prefix1 * 10 + x, prefix2 * 10 + y) * |x − y|
            sol = sol + f(prefix1 * 10 + x, prefix2 * 10 + y)

    return sol
```

The only thing left to notice is that we do not need to know the whole prefix in order to know whether we can add digit x. It is sufficient to know how many times we have added a digit before this one and has the prefix ever been different than the upper and lower interval bounds. This gives us a dynamical programming approach with O(number length * number of possible digits^2) complexity.

For additional details, check out the source code.

Necessary skills: dynamic programming

Category: dynamic programming

COCI 2013/2014	**Task ODAŠILJAČI**
3rd round, December 7th, 2013	**Author:** Matija Milišić, Antun Razum

Let us examine a part of the city between a pair of buildings and mark it with [X, Y]. To begin with, we do not care about the transmitters to the right of that interval. Every transmitter covers a part [Z, Y], while the left part [X, Z] stays uncovered. It is sufficient to find a transmitter with the minimal Z, let's call it $\mathbf{Z_L}$.

We conclude similarly for the transmitters to the right of that interval. Then the part [X, Z] is going to be covered, and the part [Z, Y] uncovered. Here we need to find the transmitter with the maximal Z, let's call it $\mathbf{Z_D}$.

When we find these two coordinates, Z_L and Z_D, for an interval [X, Y], we can calculate the coverage of an interval as: $Y - X - \max\{0, Z_L - Z_D\}$. The task is now unfolded into **two parts**: the first will calculate Z_L for every interval, and the second Z_D in a similar way.

Now we will explain how to calculate Z_L. We can notice that for every transmitter it is sufficient to observe only the part of signal located **to the right of it**.

We will sweep through all the buildings from **left to right**. By doing so, we will maintain a structure which will contain some of the transmitters from the buildings we have already swept through. The transmitters in the structure will be sorted in **ascending order according to their abscissa**. In the structure, each transmitter is paired with a point on the X axis which marks the place where the coverage of that transmitter begins, if we take into account the buildings we have swept so far (the coverage extends from that point to the right).

Now we will demonstrate an important property of this structure. For a transmitter **O**, we mark $\mathbf{X_O}$ as its abscissa, $\mathbf{H_O}$ as its height and $\mathbf{T_O}$ as its paired point, more specifically the abscissa of that point. Let us also define the predicate **better**. For a pair of transmitters A and B we say that A is **better** than B if $T_A < T_B$. Let us assume that there are two transmitters in the structure: A and B, with $X_A < X_B$ being true. If $H_A <= H_B$ is true, transmitter B can be left out because A is better. Let us assume, therefore, additionally that $H_A > H_B$ is true. If $T_A < T_B$, transmitter B can be left out, because A will always be better.

Taking this property into consideration, we are left with the following consequences: **the heights** of the transmitters in the structure are going to be **descending** and the abscissa **of the paired points** are going to be **ascending**.

Now we can easily come up with an algorithm which is applied to each building we come across (while sweeping through them from left to right). Initially, we leave out all transmitters from the end of the structure which are of equal or lower height than the current building. If the current building has a transmitter on it, we add it to the structure and Z_L for this interval is equal to X (the abscissa of the current building's transmitter). If the current building doesn't have a transmitter, we observe **the last two** transmitters in the structure. Now we **update** their paired points (maybe they will be changed after the addition of the last building). If, after the update, the last transmitter **doesn't comply with** the structure's property ($T_A > T_B$, where A is the second-to-last, and B the last transmitter in the structure), we **leave it out** of the structure. This procedure is **repeated** while possible (in other words, until the last transmitter complies with the structure's property) while always updating the paired point of the second-to-last transmitter.

We can notice that we won't need to update the paired points of the other transmitters in the structure after we've finished this procedure for a building because they do not change. Let us assume the contrary, we have two transmitters: A and B, $X_A < X_B$, $H_A > H_B$ and $T_A >= T_B$, and A needs to be updated and B does not. We construct a line PA through points (X_A, H_A) and (T_A, 0) and a line PB through points (X_B, H_B) and (T_B, 0). The last condition means that the current wall intersects with P_A and doesn't intersect with P_B. However, that is impossible because then B would not comply with the structure's property. More specifically, the condition $T_A >= T_B$ would not be met.

When we sweep through all buildings from left to right, a very similar procedure is repeated in the same way on the opposite side and this gives us arrays ZL and ZD from which we construct the final solution.

In this task's solution, we sweep through all buildings twice, maintaining the structure which can be implemented as a stack. Every step of going through the buildings is of constant complexity because when sweeping through all buildings, we insert and leave out each transmitter exactly once. Given the fact that all the structure's operations are of linear complexity, the final solution's complexity is also linear, **O(n)**, where n is the number of buildings.

Necessary skills: mathematics, amortized analysis
Category: ad hoc