AGRA - Árboles y Grafos, 2018-2
Tarea 1: Semanas 1 y 2

Para entregar el viernes 3 de agosto/domingo 5 de agosto de 2018

Problemas conceptuales a las 16:00 (3 de agosto) en el Departamento de Electrónica y Ciencias de la Computación

Problemas prácticos a las 23:59 (5 de agosto) en la arena de programación

---

Tanto los ejercicios como los problemas deben ser resueltos, pero únicamente las soluciones de los problemas deben ser entregadas. La intención de los ejercicios es entrenarlo para que domine el material del curso; a pesar de que no debe entregar soluciones a los ejercicios, usted es responsable del material cubierto en ellos.

Instrucciones para la entrega

Para esta tarea y todas las tareas futuras, la entrega de soluciones es *individual*. Por favor escriba claramente su nombre, código de estudiante y sección en cada hoja impresa entregada o en cada archivo de código (a modo de comentario). Adicionalmente, agregue la información de fecha y nombres de compañeros con los que colaboró; igualmente cite cualquier fuente de información que utilizó.

¿Cómo describir un algoritmo?

En algunos ejercicios y problemas se pide "dar un algoritmo" para resolver un problema. Una solución debe tomar la forma de un pequeño ensayo (es decir, un par de párrafos). En particular, una solución debe resumir en un párrafo el problema y cuáles son los resultados de la solución. Además, se deben incluir párrafos con la siguiente información:

- una descripción del algoritmo en castellano y, si es útil, pseudo-código;
- por lo menos un diagrama o ejemplo que muestre cómo funciona el algoritmo;
- una demostración de la corrección del algoritmo; y
- un análisis de la complejidad temporal del algoritmo.

Recuerde que su objetivo es comunicar claramente un algoritmo. Las soluciones algorítmicas correctas y descritas *claramente* recibirán alta calificación; soluciones complejas, obtusas o mal presentadas recibirán baja calificación.

Ejercicios

1.1-2 (página 11), 1.2-2, 1.2-3, 1-1 (página 14), 2.1-1, 2.1-2, 2.1-3 (página 22), 2.2-1, 2.2-2 (página 29), 2.3-1 (página 37), 2.3-4, 3.1-1 (página 52), 3.1-4, 3.1-8 (página 53), 3.2-1, 3.2-7, 3-2 columnas O, Ω y Θ (página 61), 3-3 parte a (página 62).

Problemas conceptuales

1. Escribir el código de honor del curso.
2. Ejercicios 1, 3 y 5: Orden asintótico y eficiencia algorítmica (Kleinberg & Tardos páginas 67 y 68).
3. Ejercicio 8: *Safe rungs* (Kleinberg & Tardos página 69)
4. Ejercicio 4: *Flipping Pancakes* (Erickson, Capítulo 1)
5. Ejercicio 14: *Inversions* (Erickson, Capítulo 1)
6. Ejercicios 4a, 4b y 13: Demostraciones por inducción (Erickson, Apéndice I.1).

Problemas prácticos

Hay cuatro problemas prácticos cuyos enunciados aparecen a partir de la siguiente página.

A - The Stern-Brocot Number System

Source file name: `brocot.py`
Time limit: 1 second

The *Stern-Brocot tree* is a beautiful way for constructing the set of all nonnegative fractions $\frac{m}{n}$ where m and n are relatively prime. The idea is to start with two fractions and then repeat the following operation as many times as desired:

insert $\frac{m+m'}{n+n'}$ between two adjacent fractions $\frac{m}{n}$ and $\frac{m'}{n'}$

For example, the first step gives us one new entry between $\frac{0}{1}$ and $\frac{1}{0}$:

$$\frac{0}{1}, \frac{1}{1}, \frac{1}{0};$$

and the next gives two more:

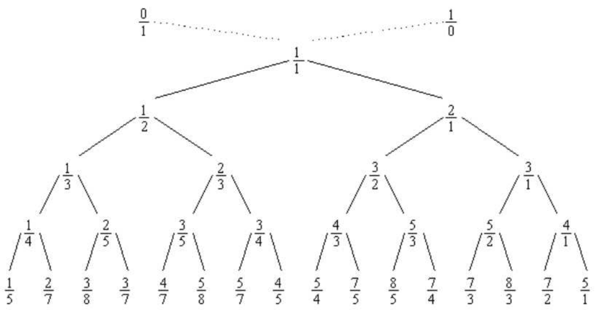
$$\frac{0}{1}, \frac{1}{2}, \frac{1}{1}, \frac{2}{1}, \frac{1}{0}.$$

The next gives four more,

$$\frac{0}{1}, \frac{1}{3}, \frac{1}{2}, \frac{2}{3}, \frac{1}{1}, \frac{3}{2}, \frac{2}{1}, \frac{3}{1}, \frac{1}{0};$$

and then we will get 8, 16, and so on.

The entire array can be regarded as an infinite binary tree structure whose top levels look like this:



The construction preserves order, and we couldn't possibly get the same fraction in two different places.

We can, in fact, regard the Stern-Brocot tree as a number system for representing rational numbers, because each positive, reduced fraction occurs exactly once. Let's use the letters *L* and *R* to stand for going down to the left or right branch as we proceed from the root of the tree to a particular fraction; then a string of *L*'s and *R*'s uniquely identifies a place in the tree.

For example, *LRRL* means that we go left from $\frac{1}{1}$ down to $\frac{1}{2}$, then right to $\frac{2}{3}$, then right to $\frac{3}{4}$, then left to $\frac{5}{7}$. We can consider *LRRL* to be a representation of $\frac{5}{7}$. Every positive fraction gets represented in this way as a unique string of *L*'s and *R*'s.

Well, actually there's a slight problem: The fraction $\frac{1}{1}$ corresponds to the empty string, and we need a notation for that. Let's agree to call it *I*, because that looks something like 1 and it stands for "identity".

In this problem, given a positive rational fraction, you are expected to represent it in Stern-Brocot number system.

Input

The input file contains multiple test cases. Each test case consists of a line contains two positive integers m and n where m and n are relatively prime. The input terminates with a test case containing two 1's for m and n, and this case must not be processed.

The input must be read from standard input.

Output

For each test case in the input file output a line containing the representation of the given fraction in the Stern-Brocot number system.

The output must be written to standard output.

Sample Input	Sample Output
5 7	LRRL
878 323	RRLRRLRLLLLRLRRR
1 1	

B - Electric Bill

Source file name: `bill.py`
Time limit: 1 second

It's year 2100. Electricity has become very expensive. Recently, your electricity company raised the power rates once more. The table below shows the new rates (consumption is always a positive integer):

Range (Crazy-Watt-hour)	**Price (Americus)**
$1 \sim 100$	2
$101 \sim 10000$	3
$10001 \sim 1000000$	5
> 1000000	7

This means that, when calculating the amount to pay, the first 100 CWh have a price of 2 Americus each; the next 9900 CWh (between 101 and 10000) have a price of 3 Americus each and so on. For instance, if you consume 10123 CWh you will have to pay $2 \times 100 + 3 \times 9900 + 5 \times 123 = 30515$ Americus.

The evil mathematicians from the company have found a way to gain even more money. Instead of telling you how much energy you have consumed and how much you have to pay, they will show you two numbers related to yourself and to a random neighbor:

A**:** the total amount to pay if your consumptions were billed together; and

B**:** the absolute value of the difference between the amounts of your bills.

If you can't figure out how much you have to pay, you must pay another 100 Americus for such a "service". You are very economical, and therefore you are sure you cannot possibly consume more than any of your neighbors. So, being smart, you know you can compute how much you have to pay. For example, suppose the company informed you the following two numbers: $A = 1100$ and $B = 300$. Then you and your neighbor's consumptions had to be 150 CWh and 250 CWh respectively. The total consumption is 400 CWh and then A is $2 \times 100 + 3 \times 300 = 1100$. You have to pay $2 \times 100 + 3 \times 50 = 350$ Americus, while your neighbor must pay $2 \times 100 + 3 \times 150 = 650$ Americus, so B is $|350 - 650| = 300$.

Not willing to pay the additional fee, you decided to write a computer program to find out how much you have to pay.

Input

The input contains several test cases. Each test case is composed of a single line, containing two integers A and B, separated by a single space, representing the numbers shown to you ($1 \leq A, B \leq 10^9$). You may assume there is always a unique solution, that is, there exists exactly one pair of consumptions that produces those numbers. The last test case is followed by a line containing two zeros separated by a single space.

The input must be read from standard input.

Output

For each test case in the input, your program must print a single line containing one integer, representing the amount you have to pay.

The output must be written to standard output.

Sample Input	Sample Output

1100 300
35515 27615
0 0
	350
	2900

C - N + NOD(N)

Source file name: `nod.py`
Time limit: 3 seconds

Consider an integer sequence N where,

$$N_0 = 1$$
$$N_i = N_{i-1} + \text{NOD}(N_{i-1}) \qquad \text{, for } i > 0.$$

Here, NOD(x) indicates the number of divisors of x. For example, NOD(100) = 9 since the divisors of 100 are $1, 2, 4, 5, 10, 20, 25, 50, 100$. Therefore, the first few terms of the N sequence are $1, 2, 4, 7, 9, 12, 18$.

Given two integers A and B, find out the number of integers in the above sequence that lie within the range $[A, B]$.

Input

The first line of input is an integer T ($T \geq 0$) that indicates the number of test cases. Each case contains two integers, A followed by B ($1 \leq A \leq B \leq 1\,000\,000$).

The input must be read from standard input.

Output

For each case, output the case number first followed by the required result.

The output must be written to standard output.

Sample Input	Sample Output

3
1 18
1 100
3000 4000
	Case 1: 7
	Case 2: 20
	Case 3: 87

D - Ultra-QuickSort

Source file name: `ultra.py`
Time limit: 2 seconds

In this problem, you have to analyze a particular sorting algorithm. The algorithm processes a sequence of n distinct integers by swapping two adjacent sequence elements until the sequence is sorted in ascending order. For the input sequence

```
9 1 0 5 4
```

Ultra-QuickSort produces the output

```
0 1 4 5 9
```

Your task is to determine how many swap operations Ultra-QuickSort needs to perform in order to sort a given input sequence.

Input

The input contains several test cases. Every test case begins with a line that contains a single integer $n < 50000$, the length of the input sequence. Each of the the following n lines contains a single integer $0 \leq a[i] < 10^{10}$, the i-th input sequence element.

Input is terminated by a sequence of length $n = 0$. This sequence must not be processed.

The input must be read from standard input.

Output

For every input sequence, your program prints a single line containing the minimum number of swap operations necessary to sort the given input sequence.

The output must be written to standard output.

Sample Input	Sample Output

```
5
9
1
0
5
4
3
1
2
3
0
```
	```
	6
	0
	```