ADA - Análisis y Diseño de Algoritmos, 2020-2
Tarea 4: Semanas 8 y 9

Para entregar el viernes 16/domingo 18 de octubre de 2020

Problemas conceptuales a las 23:59 (16 de octubre) en el Departamento de Electrónica y Ciencias de la Computación

Problemas prácticos a las 23:59 (18 de octubre) 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.

Problemas conceptuales

1. Ejercicio 2-2: *Text Segmentation* (Erickson, página 93).
2. Ejercicio 2-3: *Addition chain* (Erickson, página 94).

Problemas prácticos

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

A - Bad Code

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

In his endless attempt to keep his secret documents safe, Bob has designed yet another coding system. In this system, every character is replaced by a unique positive integer value – which we call the code for a character. However, this system – as expected – is not the best in the world. So more than one plain text can result in the same encrypted string. Here are a few more notes:

- The document consists exclusively of lower-case letters.
- The code for a letter is no more than 99.
- The encrypted string does not contain more than 100 characters.
- A code may or may not be preceded by a 0 in the encrypted string (note the 2nd sample test case).

Given the codes for each character and the encrypted string, you have to find all the plain text strings in alphabetical order that produces that encrypted string.

Input

There will be several test cases. Each test case starts with an integer N, which gives the number of unique characters in the document. Each of next N lines contains a character in the letter and its code. The last line in the test case gives the encrypted string. The last test case will have $N = 0$, which will need not be processed.

The input must be read from standard input.

Output

For each test case, print the test case number and the possible plain texts for the given encrypted string. If there is more than 100 possible strings report only the first 100 strings. Print a blank line after the output for each test case.

The output must be written to standard output.

Sample Input	**Sample Output**

```
5
a 12
b 1
c 2
d 3
e 23
123
2
o 10
x 1
1010101
0
```
	```
	Case #1
	ad
	bcd
	be

	Case #2
	ooox
	ooxx
	oxox
	oxxx
	xoox
	xoxx
	xxox
	xxxx
	```

B - Garden of Eden

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

Cellular automata are mathematical idealizations of physical systems in which both space and time are discrete, and the physical quantities take on a finite set of discrete values. A cellular automaton consists of a lattice (or array), usually infinite, of discrete-valued variables. The state of such automaton is completely specified by the values of the variables at each place in the lattice. Cellular automata evolve in discrete time steps, with the value at each place (cell) being affected by the values of variables at sites in its neighborhood on the previous time step. For each automaton there is a set of rules that define its evolution.

For most cellular automata there are configurations (states) that are unreachable: no state will produce them by the application of the evolution rules. These states are called Gardens of Eden for they can only appear as initial states. As an example consider a trivial set of rules that evolve every cell into 0; for this automaton any state with non-zero cells is a Garden of Eden.

In general, finding the ancestor of a given state (or the non-existence of such ancestor) is a very hard, compute intensive, problem. For the sake of simplicity we will restrict the problem to 1-dimensional binary finite cellular automata. This is, the number of cells is a finite number, the cells are arranged in a linear fashion and their state will be either '`0`' or '`1`'. To further simplify the problem each cell state will depend only on its previous state and that of its immediate neighbors (the one to the left and the one to the right).

The actual arrangement of the cells will be along a circumference, so that the last cell is next to the first.

Given a circular binary cellular automaton you must find out whether a given state is a Garden of Eden or a reachable state. The cellular automaton will be described in terms of its evolution rules. For example, the table below shows the evolution rules for the automaton: $Cell = XOR(Left; Right)$.

Left $[i-1]$	Cell $[i]$	Right $[i+1]$	New State		
0	0	0	0	$0 * 2^0$	
0	0	1	1	$1 * 2^1$	
0	1	0	0	$0 * 2^2$	
0	1	1	1	$1 * 2^3$	
1	0	0	1	$1 * 2^4$	
1	0	1	0	$0 * 2^5$	
1	1	0	1	$1 * 2^6$	
1	1	1	0	$0 * 2^7$	
				90	= Automaton Identifier

Notice that, with the restrictions imposed to this problem, there are only 256 different automata. An identifier for each automaton can be generated by taking the New State vector and interpreting it as a binary number (as shown in the table). For instance, the automaton in the table has identifier 90. The *Identity* automaton (every state evolves to itself) has identifier 204.

Input

The input will consist of several test cases. Each input case will describe, in a single line, a cellular automaton and a state. The first item in the line will be the identifier of the cellular automaton you must work with. The second item in the line will be a positive integer N ($4 \leq N \leq 32$) indicating the number of cells for this test case. Finally, the third item in the line will be a state represented by a string of exactly N zeros and ones. Your program must keep reading lines until the end of the input (end of file).

The input must be read from standard input.

Output

If an input case describes a Garden of Eden you must output the string `GARDEN OF EDEN`. If the input does not describe a Garden of Eden (it is a reachable state) you must output the string `REACHABLE`.

The output for each test case must be in a different line.

The output must be written to standard output.

Sample Input	**Sample Output**
`0 4 1111`	`GARDEN OF EDEN`
`204 5 10101`	`REACHABLE`
`255 6 000000`	`GARDEN OF EDEN`
`154 16 1000000000000000`	`GARDEN OF EDEN`

C - Passwords

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

Having several accounts on several servers one has to remember many passwords. You can imagine a situation when someone forgets one of them. He/she remembers only that it consisted of words x, y and z as well as two digits: one at the very beginning and the other one at the end of the password.

Your task is to write a program which will generate all possible password on the basis of given dictionary and set of rules. For the example given above the dictionary contains three words: x, y, z, and the rule is given as '`0#0`' what stands for

$\langle$digit$\rangle\langle$word_from_the_dictionary$\rangle\langle$digit$\rangle$.

Input

First line contains a number of words in the dictionary (n). The words themselves are given in n consecutive lines. The next line contains number of rules (m). Similarly consecutive m lines contain rules. Each rule consists of characters '`#`' and '`0`' given in arbitrary order. The character '`#`' stands for word from the dictionary whilst the character '`0`' stands for a digit. Input data may contain many sets of dictionaries with rules attached to them.

You can make the following assumptions. A number of words in the dictionary is greater than 0 and smaller or equal to 100 ($0 < n \leq 100$). Length of the word is greater than 0 and smaller than 256. A word may contain characters 'A'..'Z','a'..'z','0'..'9'. A number of rules is smaller than 1 000, and a rule is shorter that 256 characters. A character '0' may occur in the rule no more than 7 times, but it has to occur at least once. The character '`#`' is not mandatory meaning there can be so such characters in the rule or not.

The input must be read from standard input.

Output

For each set 'dictionary + rules' you should output two hyphens followed by a linebreak and all matching passwords given in consecutive lines. Passwords should be sorted by rules what means that first all passwords matching the first rule and all words must be given, followed by passwords matching the second rule and all words, etc. Within set of passwords matching a word and a rule an ascending digit order must be preserved.

The output must be written to standard output.

Sample Input

```
2
root
2super
1
#0
1
admin
1
#0#
```

Sample Output

```
--
root0
root1
root2
root3
root4
root5
root6
root7
root8
root9
2super0
2super1
2super2
2super3
2super4
2super5
2super6
2super7
2super8
2super9
--
admin0admin
admin1admin
admin2admin
admin3admin
admin4admin
admin5admin
admin6admin
admin7admin
admin8admin
admin9admin
```

D - Sticks

Source file name: `sticks.py`
Time limit: x seconds

George took sticks of the same length and cut them randomly until all parts became at most 50 units long. Now he wants to return sticks to the original state, but he forgot how many sticks he had originally and how long they were originally. Please help him and design a program which computes the smallest possible original length of those sticks. All lengths expressed in units are integers greater than zero.

Input

The input file contains blocks of 2 lines. The first line contains the number of sticks parts after cutting. The second line contains the lengths of those parts separated by the space. The last line of the file contains '0'.

The input must be read from standard input.

Output

The output file contains the smallest possible length of original sticks, one per line.

The output must be written to standard output.

Sample Input	**Sample Output**
`9`	`6`
`5 2 1 5 2 1 5 2 1`	`5`
`4`	
`1 2 3 4`	
`0`	

E - Super Number

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

Don’t you think 162456723 very special? Look at the picture below if you are unable to find its speciality. ($a|b$ means ‘b is divisible by a’).

Given n, m ($0 < n < m < 30$), you are to find a m-digit positive integer X such that for every i ($n \le i \le m$), the first i digits of X is a multiple of i. If more than one such X exists, you should output the lexicographically smallest one. Note that the first digit of X should not be 0.

Input

The first line of the input contains a single integer t ($t \ge 0$), the number of test cases followed. For each case, two integers n and m are separated by a single space.

The input must be read from standard input.

Output

For each test case, print the case number and X. If no such number, print ‘-1’.

The output must be written to standard output.

Sample Input	**Sample Output**
`2`	`Case 1: 1020005640`
`1 10`	`Case 2: -1`
`3 29`	