ADA - Análisis y Diseño de Algoritmos, 2019-2
Tarea 4: Semanas 8 y 9

Para entregar el lunes 23 de septiembre/domingo 22 de septiembre de 2019

Problemas conceptuales a las 16:00 (23 de septiembre) en el Departamento de Electrónica y Ciencias de la Computación

Problemas prácticos a las 23:59 (22 de septiembre) 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

22.1-1, 22.1-2, 22.1-3 (página 592), 22.2-1, 22.2-2, 22.2-4 (página 602), 22.3-1, 22.3-2, 22.3-3, 22.3-7, 22.3-12 (páginas 610-612). 22.4-1, 22.4-2, 22.4-3, 22.4-4 (páginas 614 y 615), 23.1-1, 23.1.4 (página 629), 23.2-1, 23.2-2 (página 637).

Problemas conceptuales

1. Ejercicio 7-4: *Feedback Edges* (Erickson, página 268).
2. Ejercicio 8.2: *Negative Edges* (Erickson, página 297).

Problemas prácticos

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

A - Airports

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

The government of a certain developing nation wants to improve transportation in one of its most inaccessible areas, in an attempt to attract investment. The region consists of several important locations that must have access to an airport.

Of course, one option is to build an airport in each of these places, but it may turn out to be cheaper to build fewer airports and have roads link them to all of the other locations. Since these are long distance roads connecting major locations in the country (e.g. cities, large villages, industrial areas), all roads are two-way. Also, there may be more than one direct road possible between two areas. This is because there may be several ways to link two areas (e.g. one road tunnels through a mountain while the other goes around it etc.) with possibly differing costs.

A location is considered to have access to an airport either if it contains an airport or if it is possible to travel by road to another location from there that has an airport.

You are given the cost of building an airport and a list of possible roads between pairs of locations and their corresponding costs. The government now needs your help to decide on the cheapest way of ensuring that every location has access to an airport. The aim is to make airport access as easy as possible, so if there are several ways of getting the minimal cost, choose the one that has the most airports.

Input

The first line of input contains the integer T ($T \geq 0$), the number of test cases. The rest of the input consists of T cases. Each case starts with two integers N, M and A ($0 < N \leq 10\,000, 0 \leq M \leq 100\,000, 0 < A \leq 10\,000$) separated by white space. The expression N denotes the number of locations, M is the number of possible roads that can be built, and A is the cost of building an airport.

The following M lines each contain three integers X, Y and C ($1 \leq X, Y \leq N, 0 < C \leq 10\,000$), separated by white space. X and Y are two locations, and C is the cost of building a road between X and Y.

The input must be read from standard input.

Output

Your program should output exactly T lines, one for each case. Each line should be of the form '`Case #X: Y Z`', where X is the case number Y is the minimum cost of making roads and airports so that all locations have access to at least one airport, and Z is the number of airports to be built. As mentioned earlier, if there are several answers with minimal cost, choose the one that maximizes the number of airports.

The output must be written to standard output.

Sample Input	**Sample Output**

`2`
`4 4 100`
`1 2 10`
`4 3 12`
`4 1 41`
`2 3 23`
`5 3 1000`
`1 2 20`
`4 5 40`
`3 2 30`
	`Case #1: 145 1`
	`Case #2: 2090 2`

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.

Problem definition 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	0	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`
`204 5 10101`
`255 6 000000`
`154 16 1000000000000000`
	`GARDEN OF EDEN`
	`REACHABLE`
	`GARDEN OF EDEN`
	`GARDEN OF EDEN`

C - Mice and Maze

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

A set of laboratory mice is being trained to escape a maze. The maze is made up of cells, and each cell is connected to some other cells. However, there are obstacles in the passage between cells and therefore there is a time penalty to overcome the passage Also, some passages allow mice to go one-way, but not the other way round.

Suppose that all mice are now trained and, when placed in an arbitrary cell in the maze, take a path that leads them to the exit cell in minimum time. We are going to conduct the following experiment: a mouse is placed in each cell of the maze and a count-down timer is started. When the timer stops we count the number of mice out of the maze.

Write a program that, given a description of the maze and the time limit, predicts the number of mice that will exit the maze. Assume that there are no bottlenecks is the maze, i.e. that all cells have room for an arbitrary number of mice.

Input

The input begins with a single positive integer on a line by itself indicating the number of the cases following, each of them as described below. This line is followed by a blank line, and there is also a blank line between two consecutive inputs. The maze cells are numbered $1, 2, ..., N$, where N is the total number of cells. You can assume that $N \leq 100$. The first three input lines contain N (the number of cells in the maze), E (the number of the exit cell), and T (the starting value for the count-down timer, in some arbitrary time unit). The fourth line contains the number M of connections in the maze, and is followed by M lines, each specifying a connection with three integer numbers: two cell numbers a and b (in the range $1, .., N$) and the number of time units it takes to travel from a to b. Notice that each connection is one-way, i.e., the mice can't travel from b to a unless there is another line specifying that passage. Notice also that the time required to travel in each direction might be different.

The input must be read from standard input.

Output

For each test case, the output must follow the description below. The outputs of two consecutive cases will be separated by a blank line. The output consists of a single line with the number of mice that reached the exit cell E in at most T time units.

The output must be written to standard output.

Sample Input	**Sample Output**
1 4 2 1 8 1 2 1 1 3 1 2 1 1 2 4 1 3 1 1 3 4 1 4 2 1 4 3 1	3

D - Knuth's Permutation

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

There are some permutation generation techniques in Knuth's book "The Art of Computer Programming - Volume 1". One of the processes is as follows:

For each permutation $A_1A_2 \ldots A_{n-1}$ form n others by inserting a character n in all possible places obtaining

$$nA_1A_2 \ldots A_{n-1}, A_1nA_2 \ldots A_{n-1}, \ldots, A_1A_2 \ldots nA_{n-1}, A_1A_2 \ldots A_{n-1}n$$

For example, from the permutation 231 inserting 4 in all possible places we get 4231 2431 2341 2314.

Following this rule you have to generate all the permutation for a given set of characters. All the given characters will be different and there number will be less than 10 and they all will be alpha numerals. This process is recursive and you will have to start recursive call with the first character and keep inserting the other characters in order. The sample input and output will make this clear. Your output should exactly mach the sample output for the sample input.

Input

The input contains several lines of input. Each line will be a sequence of characters. There will be less than ten alpha numerals in each line. The input will be terminated by "End of Input".

The input must be read from standard input.

Output

For each line of input generate the permutation of those characters. The input ordering is very important for the output. That is the permutation sequence for '`abc`' and '`bca`' will not be the same. Separate each set of permutation output with a blank line.

The output must be written to standard output.

Sample Input	**Sample Output**
abc bca dcba	cba bca bac cab acb abc acb cab cba abc bac bca abcd bacd bcad bcda acbd cabd cbad cbda acdb cadb cdab cdba abdc badc bdac bdca adbc dabc dbac dbca adcb dacb dcab dcba