AGRA - Árboles y Grafos, 2018-2
Tarea 5: Semanas 13, 14 y 15

Para entregar el domingo 4 de noviembre de 2018

Problemas prácticos a las 23:59 (4 de noviembre) 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

No hay en esta tarea.

Problemas prácticos

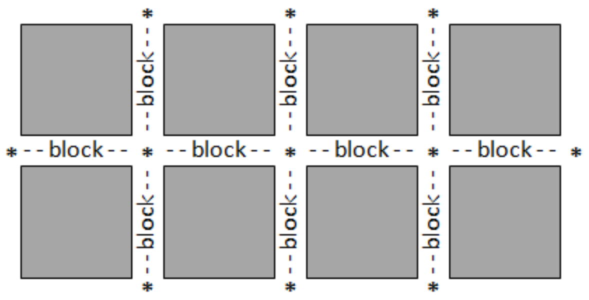
Hay cinco problemas prácticos cuyos enunciados aparecen a partir de la siguiente página.

A - Edgetown's Traffic Jams

Source file name: `jams.py`
Time limit: 4 seconds

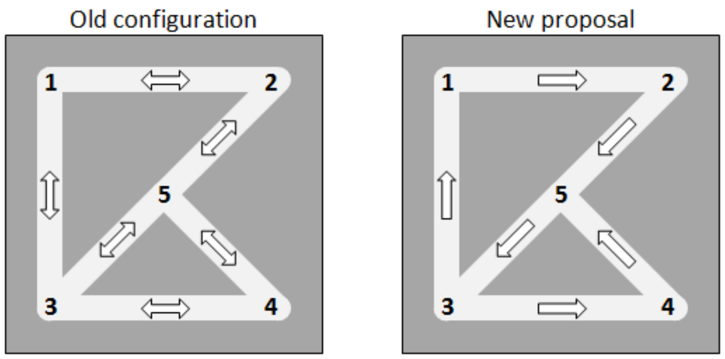
Edgetown is proudly a full-communicated city. That means that Edgetowners have decided that it must be possible to travel from any point in the city to any other point, using a car. Every point in Edgetown is located on a block, and every block connects two intersections (so that there are no intersections between a block nor two blocks connecting the same intersections). The city authorities have traditionally warranted this feature ensuring that the city plan is connected (i.e., there are not isolated intersections) and that some blocks are two-way.

Given two intersections X and Y in Edgetown, the *distance* from X to Y is measured as the minimum number of blocks that should be traveled going from X to Y. The following diagram shows a possible configuration with eight blocks and eleven intersections (marked with asterisks).



Lately there have been traffic jams at several points, almost at all times. Experts recommend a simple solution: just change some two-way blocks to be one-way blocks. However, it is clear that this should be done carefully, because accessibility among city points may be lost. On the other hand, even if accessibility is guaranteed, it is possible that distances between specific intersections may be significantly augmented.

After a lot of discussions, the Mayor's advisers have recommended to accept any proposal that increases the distance between any two intersections by a factor A augmented by a constant B, with respect to the old configuration (i.e., if the actual distance from one intersection to another is x, then the new distance should be at most $A \cdot x + B$).



You are hired to develop a program to decide if a given proposal to orient city blocks satisfies the requirements.

Input

There are several cases to analyze. Each case is described with a set of lines:

- The first line contains a non-negative integer n ($3 \le n \le 100$) that represents the number of intersections in Edgetown. Suppose that the intersections are identified by natural numbers in the set $\{1, \ldots, n\}$.
- The line $i + 1$ ($1 \le i \le n$) begins with the number i and follows with a list of intersection numbers different from i (without repetitions). That means that the intersection i is connected by a block to each of the intersections numbered by elements in the list.

- The next n lines describe, with the same already specified format, the new proposal. In the description of the blocks in the proposal should be understood that the blocks are oriented going out from the first element in the line to each of the adjacent elements (the same fact applies to the old configuration).
- The case description ends with a line with two integer values A and B ($0 \leq A \leq 10$, $0 \leq B \leq 10$).

The last test case is followed by a line containing a single 0.

The input must be read from standard input.

Output

For each case print one line with the word "`Yes`" if the proposal satisfies the given requirements, or the word "`No`" otherwise. Answers should be left aligned.

The output must be written to standard output.

Sample Input	**Sample Output**

```
5
1 2 3
2 1 5
3 4 5 1
4 3 5
5 2 3 4
1 2
2 5
3 1 4
4 5
5 3
1 2
5
1 2 3
2 1 5
3 4 5 1
4 3 5
5 2 3 4
1 2
2 5
3 1 4
4 5
5 3
2 0
3
1 2
2 1 3
3 1 2
1 2
2 3
3 1
0 2
0
```
	```
	Yes
	No
	Yes
	```

B - Interstellar Travel

Source file name: `travel.py`
Time limit: 4 seconds

The *Agency for Cross-Constellation and Interstellar Space Travel* (ACIS) is ready to offer its clients space travel among several planets across the universe.

ACIS offers a list of flight options consisting of an origin planet, a destination planet, a cost, and a duration. One of the "killer" features ACIS will offer to its clients is that of being able to plan a trip between two planets under the constraint of a maximum number of stops. That is, given a natural number n, ACIS would like to offer each client the cheapest possible trip from an origin planet to a destination planet with at most n stops. Since interstellar in-flight sleep is not pleasant, it is also important to minimize the amount of time spent in a trip.

Can you help ACIS in finding an efficient algorithm for such a task?

Input

The input consists of several test cases. Each test case begins with a line with three blank-separated integers p, f, and q ($1 \leq p \leq 300$, $0 \leq f \leq 5000$, and $0 \leq q \leq 1000$), indicating the number of planets, flights, and queries, respectively. The next p lines each contains a planet name s ($1 \leq |s| \leq 30$). The next f lines each contains two planet names and two integers s_o, s_d, c, and t (separated by a blank), denoting that there is a direct flight from s_o to s_d costing c dollars ($0 \leq c \leq 10^5$) with a duration of t units of time ($0 \leq t \leq 10^5$). The next line contains a planet name s_i indicating the initial planet for the trip. The next q lines each contains a query with a destination planet name s_f for the trip and a natural number n, both separated by a blank ($0 \leq n \leq 300$). You can assume that planet names consist only of alphabetic characters, and that s_o, s_d, s_i, and s_f are in the list of p planet names.

The input must be read from standard input.

Output

For each query s_i, s_f, n output two blank-separated integers indicating the minimum cost and the corresponding minimum travel time for this cost of an interstellar trip from s_i to s_f with at most n stops. If this is not possible, then print two blank-separated asterisks ('`*`').

Print a line with a single period ('`.`') between consecutive test cases.

The output must be written to standard output.

Sample Input

```
2 3 1
Earth
Mars
Earth Mars 2 3
Earth Mars 4 1
Earth Earth 3 2
Earth
Mars 0
3 3 5
Tatooine
Endor
Geonosis
Tatooine Endor 300 15
Endor Geonosis 10 78
Geonosis Tatooine 1 1
Endor
Endor 0
Geonosis 0
Geonosis 4
Tatooine 0
Tatooine 1
5 5 8
Earth
Kaishin
Namek
Vegeta
NewNamek
Earth Kaishin 10 10
Kaishin Namek 10 5
Kaishin Vegeta 15 30
Earth Vegeta 25 50
NewNamek Earth 100 1
Earth
Kaishin 0
Kaishin 1
Kaishin 2
Namek 0
Namek 1
Vegeta 0
Vegeta 1
NewNamek 5
```

Sample Output

```
2 3
.
0 0
10 78
10 78
* *
11 79
.
10 10
10 10
10 10
* *
20 15
25 50
25 40
* *
```

C - Interval Product

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

It's normal to feel worried and tense the day before a programming contest. To relax, you went out for a drink with some friends in a nearby pub. To keep your mind sharp for the next day, you decided to play the following game. To start, your friends will give you a sequence of N integers $X_1, X_2, \cdots, X_N$. Then, there will be K rounds; at each round, your friends will issue a command, which can be:

- a *change command*, when your friends want to change one of the values in the sequence; or
- a *product command*, when your friends give you two values I, J and ask you if the product $X_I \times X_{I+1} \times \cdots \times X_J$ is positive, negative, or zero.

Since you are at a pub, it was decided that the penalty for a wrong answer is to drink a pint of beer. You are worried this could affect you negatively at the next day's contest and you don't want to check if Ballmer's peak theory is correct. Fortunately, your friends gave you the right to use your notebook. Since you trust more your coding skills than your math, you decided to write a program to help you in the game.

Input

Each test case is described using several lines. The first line contains two integers N and K, indicating respectively the number of elements in the sequence and the number of rounds of the game ($1 \leq N, K \leq 10^5$). The second line contains N integers X_i that represent the initial values of the sequence ($-100 \leq X_i \leq 100$, for $i = 1, \ldots, N$). Each of the next K lines describes a command and starts with an uppercase letter that is either '`C`' or '`P`'. If the letter is '`C`', the line describes a change command, and the letter is followed by two integers I and V indicating that X_I must receive the value V ($1 \leq I \leq N$) and ($-100 \leq V \leq 100$). If the letter is '`P`', the line describes a product command, and the letter is followed by two integers I and J indicating that the product from X_I to X_J, inclusive must be calculated ($1 \leq I \leq J \leq N$). Within each test case there is at least one product command.

The input must be read from standard input.

Output

For each test case output a line with a string representing the result of all the product commands in the test case. The i-th character of the string represents the result of the i-th product command. If the result of the command is positive the character must be '`+`' (plus), if the result is negative the character must be '`-`' (minus), and if the result is zero the character must be '`0`' (zero).

The output must be written to standard output.

Sample Input	**Sample Output**

```
4 6
-2 6 0 -1
C 1 10
P 1 4
C 3 7
P 2 2
C 4 -5
P 1 4
5 9
1 5 -2 4 3
P 1 2
P 1 5
C 4 -5
P 1 5
P 4 5
C 3 0
P 1 5
C 4 -5
C 4 -5
```
	```
	0+-
	+-+-0
	```

D - DNA Sequencing

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

A DNA molecule consists of two strands that wrap around each other to resemble a twisted ladder whose sides, made of sugar and phosphate molecules, are connected by rungs of nitrogen-containing chemicals called bases. Each strand is a linear arrangement of repeating similar units called nucleotides, which are each composed of one sugar, one phosphate, and a nitrogenous base.

Four different bases are present in DNA: adenine (A), thymine (T), cytosine (C), and guanine (G). The particular order of the bases arranged along the sugar-phosphate backbone is called the DNA sequence; the sequence specifies the exact genetic instructions required to create a particular organism with its own unique traits.

Geneticists often compare DNA strands and are interested in finding the longest common base sequence in the two strands. Note that these strands can be represented as strings consisting of the letters a, t, c, and g. So, the longest common sequence in the two strands *atgc* and *tga* is tg. It is entirely possible that two different common sequences exist that are the same length and are the longest possible common sequences. For example in the strands *atgc* and *gctg*, the longest common sequences are *gc* and *tg*.

Write a program that accepts as input two strings representing DNA strands, and prints as output the longest common sequence(s) in lexicographical order.

Input

The input consists of several test cases. Each test cases consists of a pair of strings representing DNA strands, each on a single line. The strings are at most 300 characters long. Two consecutive test cases are separated by a blank line.

The input must be read from standard input.

Output

For each test case output the longest common sequence(s) in lexicographical order, one per line. If there isn't any common sequence between the two strings, just print:

```
No common sequence.
```

There must be a blank line between the output of two consecutive test cases.

The output must be written to standard output.

Sample Input	Sample Output
atgc tga atgc gctg	tg gc tg

E - Emacs Plugin

Source file name: `emacs.py`
Time limit: 6 seconds

Emacs is a text editor characterized by its extensibility via plugins. Sometimes Emacs is described as "the extensible, customizable, self-documenting, real-time display editor". Nowadays, Emacs is one of the main contenders in the traditional editor wars of Unix culture.

As a seasoned veteran in the competitive programming scene, Emacs is most of the time your editor of choice. You wish you could always use it but this is not possible because of a feature that is currently not working as expected: a very specific text search capability. In particular, you have gathered proof that searching a text against a regular expression pattern takes a considerable amount of time for some special kinds of patterns. As usual, to the extents of your ego, you are certain that you can implement a much better specific purpose algorithm than the general purpose one currently shipping with Emacs.

A *text* is a string made from lowercase letters ('`a`' to '`z`'). A *pattern* is a string made from lowercase letters and the wildcard symbol '`*`'. The pattern p *matches* a text t iff p can be found in t as a substring, with each symbol '`*`' in p replaced with an arbitrary substring of t.

For example, let t = "`heyhelloyou`", p_1 = "`hel*`", p_2 = "`*o*e`", and p_3 = "`e*o`". Note that p_1 matches t with '`*`' replaced, for example, with the substring "`loyo`" or even the empty string "". Also, p_3 matches t with, for example, witness "`lloy`". However, p_2 does not match t.

You have decided to implement a new algorithm for matching a pattern to a text as part of a new Emacs plugin. Can you come up with a very efficient algorithm?

Input

The input consists of several test cases. Each test case begins with an integer number n indicating the number of patterns on a single line ($1 \leq n \leq 50$). The next line contains the text t ($1 \leq |t| \leq 10^5$). The next n lines contain the patterns $p_1, p_2, \ldots, p_n$ ($1 \leq |p_i| \leq 10^5$ for each $1 \leq i \leq n$), each on a single line.

The input must be read from standard input.

Output

For each test case output n lines, where the i-th line indicates whether p_i matches t or not. If p_i matches t, then output "`yes`"; otherwise, output "`no`".

The output must be written to standard output.

Sample Input	Sample Output

```
4
heyhelloyou
hel*
*o*e
e*o
hello
1
hello
x
```
	```
	yes
	no
	yes
	yes
	no
	```