CEOI 2026

Day 1

(English)

Birdwatchers

Time limit: 10 s *Memory limit: 512 MiB*

The Birdwatchers' Society of San Serriffe has a curiously bloated and ever-changing internal structure. The Society consists of n *chapters*, and each member of the Society belongs to exactly one chapter. The chapters are numbered from 1 to n and the i'th chapter has m_i members. Thus the Society has a total of $M = m_1 + m_2 + \ldots + m_n$ members.

Each chapter is led by one of its members, who in this role is called the *officer* of the chapter. The officers are numbered the same as the chapters, so that (for each $i = 1, \ldots, n$) officer i is the one in charge of chapter i.

Moreover, the officers are organized hierarchically through a system of mentorship: each officer except one has a *mentor*, who is the officer of some other chapter. The only officer without a mentor is the *President* of the Society. If officer a is the mentor of officer b, we also say that officer b is a *disciple* of officer a. No officer is, directly or indirectly, a mentor of themselves; thus, by following the sequence from an officer to their mentor, their mentor's mentor, and so on, we eventually always reach the President.

We define the *influence* of an officer as the sum of the number of members in his chapter and of the influences of all his disciples (if he has any). It will be easily seen that the officer with the highest influence is the President, whose influence always equals M. An officer is called a *senior officer* if his influence is $\geq M/2$.

The Bylaws of the Society specify that the senior officer with the lowest influence (amongst all the senior officers) shall act as the *Treasurer* of the Society.

From time to time, an officer (other than the President) may *change his allegiance*, so that he is thenceforth the disciple of a different mentor than before (provided that his new mentor is not one of his disciples, or his disciples' disciples, etc.). Because of this, it can happen that the influence of some officers changes and the role of Treasurer falls to a different officer than before.

Task

Write a program that reads the description of the initial state of the Society and a sequence of changes of allegiance. Your program must print who the Treasurer is in the initial state of the Society as well as after each change of allegiance.

Input

The first line contains two integers, n and q, separated by a space; n is the number of chapters, q is the number of changes of allegiance.

The next n lines describe the initial state of the Society. The i'th of these lines contains two integers, s_i and m_i, separated by a space; s_i is the mentor of officer i (i.e. of the officer in charge of chapter i), while m_i is the number of members of chapter i. The value $s_i = 0$ indicates that officer i is the President of the Society and thus has no mentor.

The remaining q lines describe the changes of allegiance. The j'th of these lines contains two integers, $\hat{x}_j$ and $\hat{z}_j$, separated by a space. The meaning of these integers is as follows. Let us denote by t_j (for $j = 0, \ldots, q$) the Treasurer after the first j changes of allegiance (thus t_0 is the initial Treasurer before the first change of allegiance). Then the j'th change of allegiance consists of officer z_j becoming the new mentor of officer x_j,

where $x_j = 1 + ((t_{j-1} + \hat{x}_j) \bmod n)$ and $z_j = 1 + ((t_{j-1} + \hat{z}_j) \bmod n)$. The purpose of this representation of the values x_j and z_j is to force your program to process the changes of allegiance in the order in which they appear.

The changes of allegiance in the input data will always be valid, i.e. z_j will not be equal to x_j, nor will z_j be a disciple of x_j, a disciple of a disciple of x_j, etc. It is, however, possible that z_j was already the mentor of x_j just before the j'th change (so that nothing actually changes at that point).

Note that if your program calculates the wrong result t_j at some point, it will decode the subsequent inputs $\hat{x}_{j+1}, \hat{z}_{j+1}$, etc., incorrectly as well, and might crash with an RTE (runtime error) verdict instead of a WA (wrong answer) verdict because the incorrectly decoded inputs might be invalid (e.g. it might erroneously obtain a z_{j+1} which is a disciple of x_{j+1}).

Constraints

- $1 \leq n \leq 1\,000\,000$
- $1 \leq q \leq 30\,000$
- $1 \leq m_i$ for each $i = 1, \ldots, n$
- $m_1 + m_2 + \ldots + m_n \leq 10^9$
- $1 \leq \hat{x}_j \leq n$ and $1 \leq \hat{z}_j \leq n$ for each $j = 1, \ldots, q$.

Subtasks

- Subtask 1 (15 points): $n \leq 100$
- Subtask 2 (10 points): $n \leq 1000$
- Subtask 3 (50 points): $n \leq 300\,000$
- Subtask 4 (25 points): No additional constraints.

Output

Print the numbers $t_0, t_1, \ldots, t_q$, each on its own line, where t_j is the Treasurer after the first j changes of allegiance. Naturally, each t_j must be an integer from the range $1 \leq t_j \leq n$.

Example

Input	Output
7 2 0 1 1 3 1 3 2 3 2 1 5 2 5 1 3 7 2 7	2 2 3

Comment

Initially, officer 2 is the Treasurer (hence $t_0 = 2$). In the first change of allegiance, we read $\hat{x}_1 = 3$ and $\hat{z}_1 = 7$ and calculate $x_1 = 1+((2+3) \bmod 7) = 6$ and $z_1 = 1+((2+7) \bmod 7) = 3$; thus, officer 3 becomes the new mentor of officer 6; officer 2 remains Treasurer (hence $t_1 = 2$). In the second change of allegiance, we read $\hat{x}_2 = 2$ and $\hat{z}_2 = 7$ and calculate $x_2 = 1 + ((2 + 2) \bmod 7) = 5$ and $z_2 = 1 + ((2 + 7) \bmod 7) = 3$; thus, officer 3 becomes the new mentor of officer 5 and also becomes the new Treasurer (hence $t_2 = 3$).

DFS

Time limit: 1 s *Memory limit: 256 MiB*

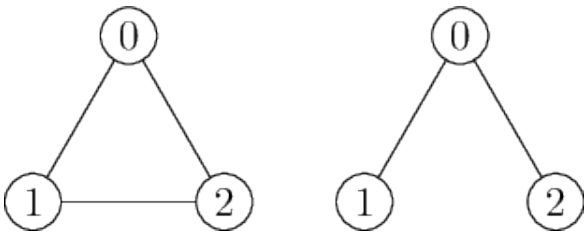
You might already be familiar with the famous DFS (depth-first search) algorithm for traversing a graph. In this problem, we will only consider connected undirected simple graphs (without loops and parallel edges) with vertices numbered 0, 1, ..., $n-1$, and the DFS algorithm will output depths and vertices as follows:

```
DFS(d, v):
    output d/v
    mark vertex v as visited
    W = the list of neighbors of v ordered by increasing numbers
    for each w in W:
        if vertex w has not yet been visited:
            DFS(d + 1, w)
```

Write a program that outputs the number of different graphs for which the call `DFS(0, n - 1)` produces the same printout as the one given on the input. For example, the output

```
0/2
1/0
2/1
```

is produced by calling `DFS(0, 2)` on either of the following two connected undirected simple 3-vertex graphs:



Input

The input is the output of the call `DFS(0, n - 1)` on an unknown n-vertex connected undirected simple graph. The input thus consists of n lines in the format `d/v`, with the first line being `0/n-1`.

Constraints

- $1 \leq n \leq 2 \cdot 10^5$

Subtasks

- Subtask 1 (10 points): $n \leq 6$.
- Subtask 2 (20 points): $n \leq 500$.
- Subtask 3 (20 points): $n \leq 10^4$.
- Subtask 4 (10 points): For each $i \in \{2, \ldots, n\}$, the i-th input line is `i-1/i-2`.
- Subtask 5 (20 points): For each $i \in \{2, \ldots, n\}$, the i-th input line is `i-1/v` for some $v \in \{0, \ldots, n-2\}$.
- Subtask 6 (20 points): No additional constraints.

Output

Print the number of different graphs with the required property. Because this number can be very large, output the result modulo 1 000 000 007.

Example

Input	Output

`0/2`
`1/0`
`2/1`
	`2`

Treasure Hunt

Time limit: 8 s *Memory limit: 256 MiB*

You are searching for a long-lost treasure on a massive $N \times N$ field of cells with a magical compass. There are a total of $K \leq 3$ treasure chests hidden at various distinct cells in the field. Your goal is simple: to find them all!

When you place the compass on one of the cells, it will find the shortest paths to a treasure chest using only moves left, up, right, and down (i.e. a closest treasure chest according to Manhattan distance). It will then show the direction of the first move. If multiple valid first moves lead to a closest treasure chest (or several of them), the compass will return all of them. If the cell contains treasure, the compass will indicate that instead.

Whenever you find a treasure chest, you empty the contents, but you cannot remove the chest – it is too heavy. Your compass does not know whether a chest is full or empty. It will point the way to a closest chest, empty or full.

Try to find the location of all treasure chests in the fewest compass queries possible.

Task

This is an interactive task. In each test case (i.e. each execution of your program), your program will have to solve several treasure hunts. You should interact with the grader by using a library provided by the organisers. This library contains the following declarations:

- **void** *NextHunt*(**int** &N, **int** &K) — call this function to start the next treasure hunt. It will set the grid size in the variable N and the number of treasures in K. If there are no more treasure hunts to be solved in the current run, the function will set N and K to -1; in that case, you should then terminate your program with the exit code 0. Note that you may call this function before finding all the treasures on the current hunt, e.g. if you are trying to only solve for partial points.

- **enum** { *TREASURE* = 0, *DIR_RIGHT* = 1, *DIR_UP* = 2, *DIR_LEFT* = 4, *DIR_DOWN* = 8 }; — these are constants used in the return values of the *Query* function (see below).

- **int** *Query*(**int** x, **int** y) — if the cell at coordinates (x, y) contains a treasure, this function returns *TREASURE*. Otherwise it returns the sum of one or more of the constants *DIR_RIGHT*, *DIR_UP*, *DIR_LEFT*, and *DIR_DOWN*, indicating which moves the compass returns when placed on the cell (x, y). The coordinates x and y must be integers from 0 to $N - 1$. (Note: in this task, y-coordinates increase from top to bottom.)

After *NextHunt* sets $N = K = -1$, your program must not call either *NextHunt* or *Query* again. It must also not call *Query* before the first call to *NextHunt*. If your program fails to observe these constraints, or if it makes more than 1000 queries within a single treasure hunt, or if it provides out-of-range values of x and/or y when calling *Query*, the library will terminate your program and return a run-time-error (RTE) verdict for the current test case.

A treasure is considered to have been found if you queried the cell containing it at least once. If your program calls *NextHunt* before finding all the treasures of the current hunt, this is not treated as an error, but it will affect your score (more on scoring below).

To access the library, your program should include the header file `treasurehuntlib.h`:

```
#include "treasurehuntlib.h"
```

You can download this header file here: `treasurehuntlib.h`.

To help you in developing your solution, a simple implementation of the library is available here: `treasurehuntlib-public.cpp`. To compile it together with your program, simply add its name as a parameter to the compiler, e.g.:

```
g++ foo.cpp treasurehuntlib-public.cpp
```

if `foo.cpp` is the name of the file containing your solution.

The implementation in `treasurehuntlib-public.cpp` supports, in addition to the above declarations, another function called **void** *InitFromFile*(**const char** ∗*fileName*), which reads a list of treasure hunts from a file and lets you play the game with those instead of generating its own random treasure hunts. You will find more details inside `treasurehuntlib-public.cpp` itself.

On the evaluation server, a different implementation of the library will be used, so you should not make any assumptions about how exactly the implementation works. You may, however, assume that it does not pollute the global namespace with any declarations other than those listed above (*NextHunt*, *Query*, and the five constants).

Your code must not read from the standard input or write to the standard output, because those will be used by our implementation of the library on the evaluation server to communicate with the rest of the evaluation environment.

Input

Constraints

- $2 \leq N \leq 10^6$
- $1 \leq K \leq 3$
- Within any single execution of your program, there will be at most 100 000 treasure hunts.
- Within any single treasure hunt, you may make at most 1000 queries.
- The grading system is not adaptive.

Subtasks

- Subtask 1 (10 points) $K = 1$
- Subtask 2 (30 points) $K = 2$
- Subtask 3 (60 points) $K = 3$.

Scoring

A subtask may consist of several test cases (several executions of your program), and each test case may consist of several treasure hunts. For the purposes of scoring, all the treasure hunts of a given subtask are evaluated together, regardless of how they were originally split amongst the test cases. For the i-th treasure hunt, let us denote the grid size by $N_i \times N_i$, the number of treasures by K_i, the number of queries made by your program by Q_i and the number of treasures found by your program by F_i. Let us furthermore denote by S the total number of points available for this subtask. Then the number of points awarded to your program for this subtask is:

- If your program always found all the treasures (i.e. if $F_i = K_i$ for all i), its score depends on $t_i = \frac{Q_i}{\lceil \log_2 N_i \rceil}$:

$$\frac{S}{2} + \frac{S}{2} \cdot \min_i f(t_i) \text{ points, where } f(t_i) = \begin{cases} 1, & t_i \leq 11 \\ 1 - (t_i - 11)/9, & 11 \leq t_i \leq 20 \\ 0, & t \geq 20. \end{cases}$$

- If your program did not always find all the treasures (i.e. there exists an i such that $F_i < K_i$), it receives

$$\frac{S}{2} \cdot \min_i \frac{F_i}{K_i} \text{ points.}$$

In other words, you get half of the points for finding all the treasures, and the other half for finding them in as few queries as possible. For a perfect score, your solution should find all the treasures in at most $11\lceil \log_2 N_i \rceil$ queries. Between $11\lceil \log_2 N_i \rceil$ and $20\lceil \log_2 N_i \rceil$ queries, the score decreases linearly; and if your solution makes more than $20\lceil \log_2 N_i \rceil$ queries, it will only get the first half of the points for finding all the treasures. (The symbols $\lceil \cdot \rceil$ mean that the value of $\log_2 N_i$ is rounded up to the nearest integer.)

If the formulas above result in a non-integer number of points for the subtask, it will be rounded to the nearest integer.

If your program has a runtime error or doesn't adhere to the aforementioned protocol in using the library, it will get 0 points for the whole subtask. To receive partial points for finding a subset of the trasures, your program therefore needs to finish the search gracefully by calling *NextHunt*.

Example

Call	Return value
NextHunt(N, K)	$N = 4$, $K = 1$
Query(2, 0)	*DIR_DOWN* + *DIR_RIGHT* = 9
Query(3, 1)	*DIR_DOWN*
Query(3, 2)	*TREASURE*
NextHunt(N, K)	$N = -1$, $K = -1$