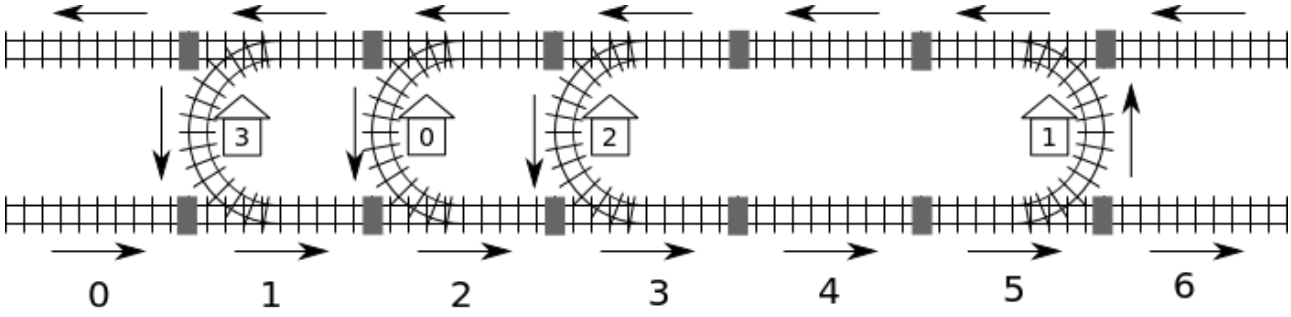
Rail

Taiwan has a big railway line connecting the western and eastern shores of the island. The line consists of m blocks. The consecutive blocks are numbered $0, \ldots, m-1$, starting from the western end. Each block has a one-way west-bound track on the north, a one-way east-bound track on the south, and optionally a train station between them.

There are three types of blocks. A type *C* block has a train station that you must enter from the northern track and exit to the southern track, a type *D* block has a train station that you must enter from the southern track and exit to the northern track, and a type *empty* block has no train station. For example, in the following figure block 0 is type empty, block 1 is type C, and block 5 is type D. Blocks connect to each other horizontally. Tracks of adjacent blocks are joined by *connectors*, shown as shaded rectangles in the following figure.



The rail system has n stations numbered from 0 to $n-1$. We assume that *we can go from any station to any other station* by following the track. For example we can go from station 0 to station 2 by starting from block 2, then passing through blocks 3 and 4 by the southern track, and then passing through station 1, then passing through block 4 by the northern track, and finally reaching station 2 at block 3.

Since there are multiple possible routes, the distance from one station to another is defined as the *minimum* number of connectors the route passes through. For example the shortest route from station 0 to 2 is through blocks 2-3-4-5-4-3 and passes through 5 connectors, so the distance is 5.

A computer system manages the rail system. Unfortunately after a power outage the computer no longer knows where the stations are and what types of blocks they are in. The only clue the computer has is the block number of station 0, which is always in a type C block. Fortunately the computer can query the distance from any station to any other station. For example, the computer can query 'what is the distance from station 0 to station 2?' and it will receive 5.

Task

You need to implement a function `findLocation` that determines for each station the block number and block type.

- `findLocation(n, first, location, stype)`
  - `n`: the number of stations.
  - `first`: the block number of station 0.
  - `location`: array of size n; you should place the block number of station i into `location[i]`.
  - `stype`: array of size n; you should place the block type of station i into `stype[i]`: 1 for type C and 2 for type D.

You can call a function `getDistance` to help you find the locations and types of stations.

- `getDistance(i, j)` returns the distance from station `i` to station `j`. `getDistance(i, i)` will return 0. `getDistance(i, j)` will return -1 if `i` or `j` is outside the range $0 \le i, j \le n - 1$.

Subtasks

In all subtasks the number of blocks m is no more than 1,000,000. In some subtasks the number of calls to `getDistance` is limited. The limit varies by subtask. Your program will receive 'wrong answer' if it exceeds this limit.

subtask	points	n	getDistance calls	note
1	8	$1 \le n \le 100$	unlimited	All stations except 0 are in type D blocks.
2	22	$1 \le n \le 100$	unlimited	All stations to the right of station 0 are in type D blocks, and all stations to the left of station 0 are in type C blocks.
3	26	$1 \le n \le 5,000$	$n(n-1)/2$	no additional limits
4	44	$1 \le n \le 5,000$	$3(n-1)$	no additional limits

Implementation details

You have to submit exactly one file, called `rail.c`, `rail.cpp` or `rail.pas`. This file implements `findLocation` as described above using the following signatures. You also need to include a header file `rail.h` for C/C++ implementation.

C/C++ program

```
void findLocation(int n, int first, int location[], int stype[]);
```

Pascal program

```
procedure findLocation(n, first : longint; var location,
stype : array of longint);
```

The signatures of `getDistance` are as follows.

C/C++ program

```
int getDistance(int i, int j);
```

Pascal program

```
function getDistance(i, j: longint): longint;
```

Sample grader

The sample grader reads the input in the following format:

- line 1: the subtask number
- line 2: `n`
- line $3+i$, ($0 \le i \le n-1$): `stype[i]` (1 for type C and 2 for type D), `location[i]`.

The sample grader will print `Correct` if `location[0]` ... `location[n-1]` and `stype[0]` ... `stype[n-1]` computed by your program match the input when `findLocation` returns, or `Incorrect` if they do not match.

Wall

Jian-Jia is building a wall by stacking bricks of the same size together. This wall consists of n columns of bricks, which are numbered 0 to $n-1$ from left to right. The columns may have different heights. The height of a column is the number of bricks in it.

Jian-Jia builds the wall as follows. Initially there are no bricks in any column. Then, Jian-Jia goes through k phases of *adding* or *removing* bricks. The building process completes when all k phases are finished. In each phase Jian-Jia is given a range of consecutive brick columns and a height h, and he does the following procedure:

- In an *adding* phase, Jian-Jia adds bricks to those columns in the given range that have less than h bricks, so that they have exactly h bricks. He does nothing on the columns having h or more bricks.
- In a *removing* phase, Jian-Jia removes bricks from those columns in the given range that have more than h bricks, so that they have exactly h bricks. He does nothing on the columns having h bricks or less.

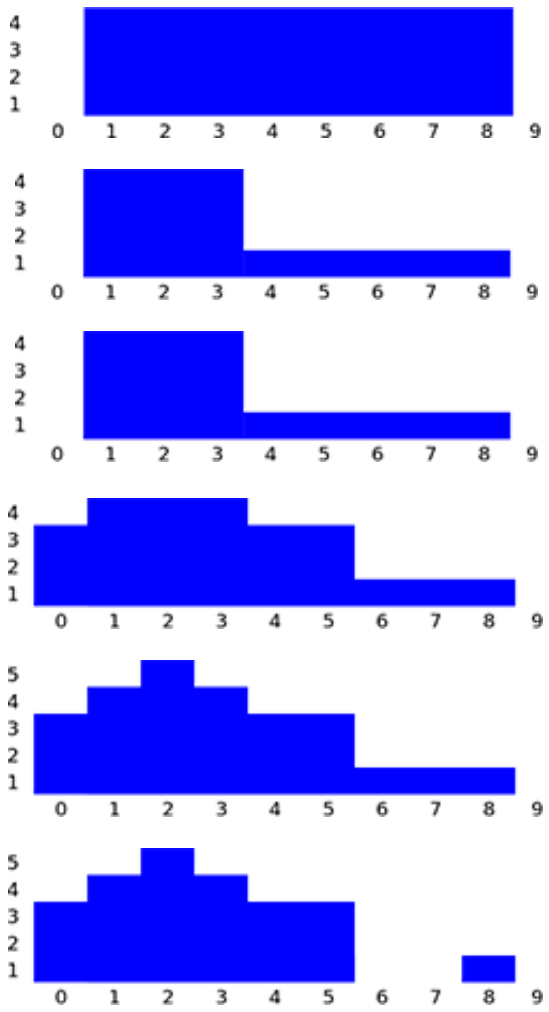
Your task is to determine the final shape of the wall.

Example

We assume that there are 10 brick columns and 6 wall building phases. All ranges in the following table are inclusive. Diagrams of the wall after each phase are shown below.

phase	type	range	height
0	add	columns 1 to 8	4
1	remove	columns 4 to 9	1
2	remove	columns 3 to 6	5
3	add	columns 0 to 5	3
4	add	column 2	5
5	remove	columns 6 to 7	0

Since all columns are initially empty, after phase 0 each of the columns 1 to 8 will have 4 bricks. Columns 0 and 9 remain empty. In phase 1, the bricks are removed from columns 4 to 8 until each of them has 1 brick, and column 9 remains empty. Columns 0 to 3, which are out of the given range, remain unchanged. Phase 2 makes no change since columns 3 to 6 do not have more than 5 bricks. After phase 3 the numbers of bricks in columns 0, 4, and 5 increase to 3. There are 5 bricks in column 2 after phase 4. Phase 5 removes all bricks from columns 6 and 7.



Task

Given the description of the k phases, please calculate the number of bricks in each column after all phases are finished. You need to implement the function `buildWall`.

- `buildWall(n, k, op, left, right, height, finalHeight)`
  - `n`: the number of columns of the wall.
  - `k`: the number of phases.
  - `op`: array of length k; `op[i]` is the type of phase i: 1 for an adding phase and 2 for a removing phase, for $0 \leq i \leq k-1$.
  - `left` and `right`: arrays of length k; the range of columns in phase i starts with column `left[i]` and ends with column `right[i]` (including both endpoints `left[i]` and `right[i]`), for $0 \leq i \leq k-1$. You will always have `left[i]` $\leq$ `right[i]`.
  - `height`: array of length k; `height[i]` is the height parameter of phase i, for $0 \leq i \leq k-1$.
  - `finalHeight`: array of length n; you should return your results by placing the final number of bricks in column i into `finalHeight[i]`, for $0 \leq i \leq n-1$.

Subtasks

For all subtasks the height parameters of all phases are nonnegative integers less or equal to $100,000$.

subtask	points	n	k	note
1	8	$1 \le n \le 10,000$	$1 \le k \le 5,000$	no additional limits
2	24	$1 \le n \le 100,000$	$1 \le k \le 500,000$	all adding phases are before all removing phases
3	29	$1 \le n \le 100,000$	$1 \le k \le 500,000$	no additional limits
4	39	$1 \le n \le 2,000,000$	$1 \le k \le 500,000$	no additional limits

Implementation details

You have to submit exactly one file, called `wall.c`, `wall.cpp` or `wall.pas`. This file implements the subprogram described above using the following signatures. You also need to include a header file `wall.h` for C/C++ program.

C/C++ program

```
void buildWall(int n, int k, int op[], int left[], int right[],
int height[], int finalHeight[]);
```

Pascal program

```
procedure buildWall(n, k : longint; op, left, right, height :
array of longint; var finalHeight : array of longint);
```

Sample grader

The sample grader reads the input in the following format:

- line 1: `n`, `k`.
- line $2+i$ ($0 \le i \le k-1$): `op[i]`, `left[i]`, `right[i]`, `height[i]`.

Game

Jian-Jia is a young boy who loves playing games. When he is asked a question, he prefers playing games rather than answering directly. Jian-Jia met his friend Mei-Yu and told her about the flight network in Taiwan. There are n cities in Taiwan (numbered 0, ..., $n-1$), some of which are connected by flights. Each flight connects two cities and can be taken in both directions.

Mei-Yu asked Jian-Jia whether it is possible to go between any two cities by plane (either directly or indirectly). Jian-Jia did not want to reveal the answer, but instead suggested to play a game. Mei-Yu can ask him questions of the form "Are cities x and y *directly* connected with a flight?", and Jian-Jia will answer such questions immediately. Mei-Yu will ask about every pair of cities exactly once, giving $r = n(n-1)/2$ questions in total. Mei-Yu wins the game if, after obtaining the answers to the first i questions for some $i < r$, she can infer whether or not it is possible to travel between every pair of cities by flights (either directly or indirectly). Otherwise, that is, if she needs all r questions, then the winner is Jian-Jia.

In order for the game to be more fun for Jian-Jia, the friends agreed that he may forget about the real Taiwanese flight network, and invent the network as the game progresses, choosing his answers based on Mei-Yu's previous questions. Your task is to help Jian-Jia win the game, by deciding how he should answer the questions.

Examples

We explain the game rules with three examples. Each example has $n = 4$ cities and $r = 6$ rounds of question and answer.

In the first example (the following table), Jian-Jia *loses* because after round 4, Mei-Yu knows for certain that one can travel between any two cities by flights, no matter how Jian-Jia answers questions 5 or 6.

round	question	answer
1	0, 1	yes
2	3, 0	yes
3	1, 2	no
4	0, 2	yes
-----	--------	------
5	3, 1	no
6	2, 3	no

In the next example Mei-Yu can prove after round 3 that no matter how Jian-Jia answers questions 4, 5, or 6, one *cannot* travel between cities 0 and 1 by flights, so Jian-Jia loses again.

round	question	answer
1	0, 3	no
2	2, 0	no
3	0, 1	no
-----	--------	------
4	1, 2	yes
5	1, 3	yes
6	2, 3	yes

In the final example Mei-Yu cannot determine whether one can travel between any two cities by flights until all six questions are answered, so Jian-Jia *wins* the game. Specifically, because Jian-Jia answered *yes* to the last question (in the following table), then it is possible to travel between any pair of cities. However, if Jian-Jia had answered *no* to the last question instead then it would be impossible.

round	question	answer
1	0, 3	no
2	1, 0	yes
3	0, 2	no
4	3, 1	yes
5	1, 2	no
6	2, 3	yes

Task

Please write a program that helps Jian-Jia win the game. Note that neither Mei-Yu nor Jian-Jia knows the strategy of each other. Mei-Yu can ask about pairs of cities in any order, and Jian-Jia must answer them immediately without knowing the future questions. You need to implement the following two functions.

- `initialize(n)` -- We will call your `initialize` first. The parameter n is the number of cities.
- `hasEdge(u, v)` -- Then we will call `hasEdge` for $r = n(n-1)/2$ times. These calls represent Mei-Yu's questions, in the order that she asks them. You must answer whether there is a direct flight between cities u and v. Specifically, the return value should be 1 if there is a direct flight, or 0 otherwise.

Subtasks

Each subtask consists of several games. You will only get points for a subtask if your program wins all of the games for Jian-Jia.

subtask	points	n
1	15	$n = 4$

subtask	points	n
2	27	$4 \leq n \leq 80$
3	58	$4 \leq n \leq 1500$

Implementation details

You have to submit exactly one file, called `game.c`, `game.cpp` or `game.pas`. This file implements the subprograms described above using the following signatures.

C/C++ programs

```
void initialize(int n);
int hasEdge(int u, int v);
```

Pascal programs

```
procedure initialize(n: longint);
function hasEdge(u, v: longint): longint;
```

Sample grader

The sample grader reads the input in the following format:

- line 1: `n`
- the following r lines: each line contains two integers `u` and `v` that describe a question regarding cities u and v.