Problem A. Snowmen

Time limit:	1 second
Memory limit:	256 MiB

It's winter. Year 2222. The news is: cloning of snowmen becomes available.

Snowman consists of zero or more snowballs put one atop another. Each snowball has some mass. Cloning of a snowman produces its exact copy.

Andrew initially has one empty snowman and performs a sequence of the following operations. Clone one of his snowmen and

- either put a new snowball on the top of the new snowman;
- or remove the topmost snowball from the new snowman (the new snoman must be nonemtpy).

He wants to know the total mass of all his snowmen after he performs all operations.

Input

The first line of input contains an integer n ($1 \le n \le 200\,000$). The following lines describe operations, the i-th operation is on of the following:

- `t m` — clone the snowman number t ($0 \le t < i$) to get the snowman number i, and put the ball with the mass m on the top of the snowman number i ($0 < m \le 1000$);
- `t 0` — clone the snowman number t ($0 \le t < i$) to get the snowman number i and remove topmost snowball. It is guaranteed that the snowman number t is not empty.

All masses are integer.

Output

Output the total mass of all snowmen in the end.

Examples

standard input	standard output

8
0 1
1 5
2 4
3 2
4 3
5 0
6 6
1 0
	74

Problem B. Persistent Queue

Time limit:	1 second
Memory limit:	256 MiB

Persistent data structures are designed to allow access and modification of any version of data structure. In this problem you are asked to implement persistent queue.

Queue is the data structure that maintains a list of integer numbers and supports two operations: push and pop. Operation push(x) adds x to the end of the list. Operation pop returns the first element of the list and removes it.

In persistent version of queue each operation takes one additional argument v. Initially the queue is said to have version 0. Consider the i-th operation on queue. If it is push(v, x), the number x is added to the end of the v-th version of queue and the resulting queue is assigned version i (the v-th version is not modified). If it is pop(v), the front number is removed from the v-th version of queue and the resulting queue is assigned version i (similarly, version v remains unchanged).

Given a sequence of operations on persistent queue, print the result of all pop operations.

Input

The first line of the input file contains n — the number of operations ($1 \le n \le 200\,000$). The following n lines describe operations. The i-th of these lines describes the i-th operation. Operation push(v, x) is described as "`1` v x", operation pop(v) is described as "`-1` v". It is guaranteed that pop is never applied to an empty queue. Elements pushed to the queue fit standard signed 32-bit integer type.

Output

For each pop operation print the element that was extracted.

Examples

standard input	standard output

10
1 0 1
1 1 2
1 2 3
1 2 4
-1 3
-1 5
-1 6
-1 4
-1 8
-1 9
	1
	2
	3
	1
	2
	4

Problem C. Persistant Array

Time limit: 1 second
Memory limit: 256 MiB

You are given the initial revision of an array. You have to perform two operations on it.

- `create` i j x ($a_{new} = a_i$; $a_{new}[j] = x$) — create the new revision from the i-th one, assign the j-th element to x, other elements remain the same as in the i-th revision.
- `get` i j (print $a_i[j]$) — report the value of the j-th element of the i-th revision.

Input

Input contains integer n ($1 \le n \le 10^5$), followed by elements of the initial revision of the array. The initial revision has number 1. The number of queries m ($1 \le m \le 10^5$) follows, then m queries. See sample input for queries formatting. The new revision of the array created when there are k revisions, get number $k+1$. All elements of the array are integers from 0 to 10^9, inclusive. Array is indexed from 1 to n, inclusive.

Output

For each `get` query output the corresponding element.

Example

standard input	standard output

```
6
1 2 3 4 5 6
11
create 1 6 10
create 2 5 8
create 1 5 30
get 1 6
get 1 5
get 2 6
get 2 5
get 3 6
get 3 5
get 4 6
get 4 5
```
	```
	6
	5
	10
	5
	10
	8
	6
	30
	```

Problem D. Persistent Multiset

Time limit:	1 second
Memory limit:	256 MiB

You have to implement partially persistent multiset (multiset is a set that can contains several copies of the same element). The multiset contains positive integers not exceeding m. It must support the following operations:

- `add` x — add x to the multiset;
- `remove` x — remove one of the elements equal to x from the multiset if there is at least one. If there are none, the multiset doesn't change.
- `different` v — output the number of different x, that are contained in revision v of the multiset;
- `unique` v — output the number of different x, such that there is exactly one x contained in revision v of the multiset;
- `count` x v — output the number of copies of x that are contained in revision v of the multiset.

Initially the multiset is empty and has revision 0. After the i-th operation it has revision i.

Input

The first line contains two integers: n, m — the number of operations to perform and the maximal possible value in the multiset ($1 \le n, m \le 200\,000$). The following n lines describe operations. To force online answers, instead of version numbers for all operations that require version you are given an integer y ($0 \le y \le n$). Let s be the sum of answers for all preceeding `different`, `unique` and `count` opearrations. The number v for the i-th operation if it is `different`, `unique` or `count`, is calculated as $v = (y+s) \bmod i$. All values x for `add`, `remove` and `count` queries are positive integers not exceeding m.

Output

Output answers for `different`, `unique` and `count` operations, one on a line.

Examples

standard input	standard output	Notes
`9 3`	`2`	*Actual queries:*
`add 2`	`1`	`add 2`
`add 1`	`2`	`add 1`
`add 2`	`0`	`add 2`
`different 3`	`1`	`different 3`
`unique 1`		`unique 3`
`remove 2`		`remove 2`
`unique 3`		`unique 6`
`count 3 1`		`count 3 6`
`count 2 1`		`count 2 6`

Problem E. Persistent Priority Queue

Time limit:	1 second
Memory limit:	256 MiB

You must implement data structure that stores a multiset (a set that can have several copies of the same element), and allows to modify any revision with one of the following operations:

1. Given v and x, add element x to the data structure revision v, and print the minimal element in the resulting multiset.
2. Given v and u, unite multisets contained in revisions v and u, after that print the minimal element in the resulting multiset.
3. Given v, print the minimal element in revision v and remove it. If the set is empty, you must report that it is empty, and the new revision remains empty.

Initially the data structure is empty and has revision 0. After the i-th operation the result of this operation has the revision i.

Input

The first line of input contains n — the number of operations.

You must answer the queries online, so you must maintain the variable s. After you answer the query and the answer is x, the new value of s is $s = (s_{old} + x) \bmod 239017$. If the answer is `empty`, the value of s doesn't change.

The following n lines contains descriptions of the operations.

Opeations of the first type are described with the line `1 a b`, where a and b are non-negative integers that describe v and x for the corresponding operation: $v = (a + s) \bmod i$, $x = (b + 17s) \bmod (10^9 + 1)$, here i is the number of the operation.

Operations of the second type are described with the line `2 a b`, where a and b are non-negative integers that describe v and u for the corresponding operation: $v = (a + s) \bmod i$, $u = (b + 13s) \bmod i$, here i is the number of the operation.

Operations of the third type are described with the line `3 a`, where a is a non-negative integer that describes v for the corresponding operation as $v = (a + s) \bmod i$, here i is the number of the operation.

The number of queries doesn't exceed 200 000. It is guaranteed that each multiset that is constructed contains at most 2^{63} elements.

Output

Output n lines, each line must contain non-negative integer of a word `empty`.

For each operation of the first or of the second type print the minimal value in the resulting multiset, or a word `empty` if it is empty.

For each operation of the third type print the minimal element in the multiset, or a word `empty`, if it is empty.

Example

standard input	standard output

9
1 0 2
1 0 999999970
2 2 0
3 0
2 4 4
3 0
3 0
3 0
3 8
	2
	3
	2
	2
	2
	2
	2
	3
	empty

Problem F. Intercity Express

Time limit:	3 seconds
Memory limit:	256 MiB

Andrew is developing the a system for train ticket sales. He is going to test it on Intercity Express line that connects two large cities and has $n - 2$ intermediate stations, so there are a total of n stations numbered from 1 to n.

Intercity Express train has s seats numbered from 1 to s. In test mode the system has access to a database that contains already sold tickets in direction from station 1 to station n and needs to answer questions whether it is possible to sell a ticket from station a to station b and if so, what is the minimal number of seat that is vacant on all segments between a and b. Initially the system will have read only access, so even if there is a vacant seat, it should report so, but should not modify the data to report it reserved.

Help Andrew to test his system by writing a program that would answer such questions.

Input

The first line of the input file contains n — the number of stations, s — the number of seats and m — the number of already sold tickets ($2 \le n \le 10^9$, $1 \le s \le 100\,000$, $0 \le m \le 100\,000$). The following m lines describe tickets, each ticket is described by c_i, a_i, and b_i — the seat that the owner of the ticket occupies, the station from which the ticket is sold, and the station to which the ticket is sold ($1 \le c_i \le s$, $1 \le a_i < b_i \le n$).

The following line contains q — the number of queries ($1 \le q \le 100\,000$). A special value p must be maintained when reading queries. Initially $p = 0$. The following $2q$ integers describe queries. Each query is described with two numbers: x_i and y_i ($x_i < y_i$). To get cities a and b between which the seat availability is requested use the following formulae: $a = x_i + p$, $b = y_i + p$. The answer to the query is 0 if there is no seat that is vacant on each segment between a and b, or the minimal number of seat that is vacant.

After answering the query, assign the answer for the query to p.

Output

For each query output the answer to it.

Example

standard input	standard output
`5 3 5`	`1`
`1 2 5`	`2`
`2 1 2`	`2`
`2 4 5`	`3`
`3 2 3`	`0`
`3 3 4`	`2`
`10`	`0`
`1 2   1 2   1 2   2 3   -2 0`	`0`
`2 4   1 3   1 4   2 5   1 5`	`0`
	`0`

Note that actual queries are $(1, 2)$, $(2, 3)$, $(3, 4)$, $(4, 5)$, $(1, 3)$, $(2, 4)$, $(3, 5)$, $(1, 4)$, $(2, 5)$, $(1, 5)$.

Problem G. Rollback

Time limit:	2 seconds
Memory limit:	256 MiB

Sergey has an array of integers $a_1, a_2, \ldots, a_n$, $1 \le a_i \le m$. He wants to answer the following questions: given l what is the minimal r such that there are at least k different values among $a_l, a_{l+1}, \ldots, a_r$.

Input

The first line of input contains two integers: n and m ($1 \le n, m \le 100\,000$). The second line cotnains n integers $a_1, a_2, \ldots, a_n$ ($1 \le a_i \le m$).

The following line contains q — the number of queries to answer. ($1 \le q \le 100\,000$). To answer the queries online you must maintain an integer p, initially $p = 0$. Each query is specified with two integers x_i and y_i, use them to get query parameters: $l_i = ((x_i + p) \bmod n) + 1$, $k_i = ((y_i + p) \bmod m) + 1$ ($1 \le l_i, x_i \le n$, $1 \le k_i, y_i \le m$). Let the answer to the i-th query be r_i. After answering the question, set p equal to r_i.

Output

For each query output the minimal value of r_i, of 0 if there is no such r_i.

Examples

standard input	standard output

7 3
1 2 1 3 1 2 1
4
7 3
7 1
7 1
2 2
	1
	4
	0
	6

Problem H. Urns and Balls

Time limit:	2 seconds
Memory limit:	256 MiB

Consider n different urns and n different balls. Initially, there is one ball in each urn.

There is a special device designed to move the balls. Using this device is simple. First, you choose some range of consecutive urns. The device then lifts all the balls form these urns. After that, you specify the destination which is another range of urns having the same length. The device then moves the lifted balls and places them in the destination urns. Each urn can contain any number of balls.

Given a sequence of movements for this device, find where will each of the balls be placed after all these movements.

Input

First line contains two integers n and m, the number of urns and the number of movements ($1 \le n \le 100\,000$, $1 \le m \le 50\,000$). Each of the next m lines contain three integers $count_i$, $from_i$ and to_i which mean that the device simultaneously moves all balls from urn $from_i$ to urn to_i, all balls from $from_i + 1$ to urn $to_i + 1$, $\ldots$, all balls from urn $from_i + count_i - 1$ to urn $to_i + count_i - 1$ ($1 \le count_i, from_i, to_i \le n$, $\max(from_i, to_i) + count_i \le n + 1$).

Output

Output exactly n numbers from 1 to n: the final positions of all balls. The first number is the final position of the ball which was initially in urn 1, the second number is the final position of the ball from urn 2, and so on.

Examples

standard input	standard output

2 3
1 1 2
1 2 1
1 2 1
	1 1

10 3
1 9 2
3 7 3
8 3 1
	1 2 1 2 3 4 1 2 2 8

Problem I. Greedy

Time limit:	1 second
Memory limit:	256 MiB

You have n objects in a given order, the i-th object has weight s_i and cost c_i. You have q knapsacks, the j-th of them can contain objects with the total weight not exceeding w_j. You want to try to greedily fill each knapsack independently and see what is the maximal cost of all objects to put there.

Take a knapsack. You try to put all objects to the knapsack, one after another in order. If you can put the current object to the knapsack, not exceeding the knapsack capacity put it there. If you cannot put the object to the knapsack, you skip it and move on to the next object.

For each knapsack output the total cost of all objects in this knapsack if they are put there using the described algorithm. All knapsacks must be considered independently, it is irrelevant for a knapsack which objects were put to other knapsacks.

Input

The first line contains n — the number of objects ($1 \le n \le 10^4$).

The second line contains n integers $s_1, s_2, \ldots, s_n$ — weights of the objects ($1 \le s_i \le 10^{13}$).

The third line n integers $c_1, c_2, \ldots, c_n$ — costs of the objects ($1 \le c_i \le 10^4$).

The fourth line contains q — the number of knapsacks to consider ($1 \le q \le 10^6$).

The fifth line contains q integers $w_1, w_2, \ldots, w_q$ — capacities of knapsacks ($1 \le w_i \le 10^{18}$).

Output

For each knapsack output q integers — the total cost of all objects put to the corresponding knapsack.

Example

standard input	standard output

5
5 3 2 4 1
1 2 3 4 5
3
4 8 100
	7
	3
	15