Classic Tree Problem

Time limit: 4 seconds
Memory limit: 1024 megabytes

This problem can only be solved using C++ or Python3.

One morning, Egor went outside and saw on the asphalt classic hopscotch squares — n squares, each containing some numbers, with the i-th square containing the number a_i. But these were not ordinary squares; each square was connected to some other squares.

The game of hopscotch involves starting from a square and jumping to another square connected to the current one, with the restriction that you cannot jump to the same square more than once. The game can end at any moment, and the sequence of visited squares $v_0, v_1, \ldots, v_{k-1}$ will be called the jump sequence.

However, the squares on the asphalt were drawn in the form of a tree, meaning there is a unique way to reach any square from another through some sequence of jumps. The "coolness" of the jump sequence is defined as the number of indices i such that $a_{v_i} = i$.

Egor wanted to find out the coolness of the jump sequences for some pairs of squares u, v, where $v_0 = u$ and $v_{k-1} = v$, to determine whether it is worth making such a sequence of jumps.

Egor wanted to show off his coolness, but he is constantly being interrupted. Therefore, he sometimes erases the number in the s-th square and writes a new number x. Since counting numbers is a difficult task for Egor, he turned to you for help.

In the next q minutes, you need to assist Egor with his questions about the coolness of the jumps. In minute i, Egor can perform one of two actions:

1. Erase the number in square s and write the number x in its place.
2. Ask about the coolness of the jump sequence from square u to square v.

You are required to answer each of Egor's questions.

Solution Format

This is an unusual problem. It has a testing format with a grader, where you only need to implement the function `solve` in your solution. This function will be called by the testing program of the jury (the grader), and the returned value of the function will be accepted as the solution to the problem.

In particular, this means that there should be **no input or output** in the code you submit. If you code in C++, your code **must not** contain a function `main`. If necessary, you can implement any number of helper functions, structures, classes, and global variables, but all the code of your solution must be in one file.

For a solution in C++, you must implement the following function:

```
std::vector<int> solve(int n, int q, std::vector<int> a, std::vector<int> p,
                       std::vector<int> qt, std::vector<int> qx, std::vector<int> qy);
```

For a solution in Python3 or Pypy3, you must implement the following function:

```
def solve(n, q, a, p, qt, qx, qy):
```

Here, n and q are integers; a and p are lists of integers of length n; qt, qx, and qy are lists of integers of length q.

Note that the jury does not guarantee the possibility of achieving full points on Python3 or Pypy3.

In both languages, the function `solve` takes the following arguments:

- n ($1 \le n \le 10^6$) — the number of squares.

- q ($1 \le q \le 10^6$) — the number of queries.
- a ($0 \le a[i] \le n$) — an array of size n, consisting of the numbers initially written in the squares.
- p ($-1 \le p[i] < n$) — an array of size n, where $p[i]$ is the parent number of square i if the hopscotch is represented as a tree. For the root of the tree, $p[i]$ will be -1. It is guaranteed that the array defines a valid tree.
- qt ($1 \le qt[i] \le 2$) — an array of size q, consisting of the types of queries.
- qx and qy — arrays defining the queries.

For each $0 \le i < q$, the i-th query is defined as follows:

- If $qt[i] = 1$, then in the i-th query, it requires erasing the number in the square with index $qx[i]$ and writing the number $qy[i]$ in its place. In this case, the following constraints hold: $0 \le qx[i] < n$, $0 \le qy[i] \le n$.
- If $qt[i] = 2$, then in the i-th query, it requires finding the coolness of the jump sequence from square $qx[i]$ to square $qy[i]$. In this case, the following constraints hold: $0 \le qx[i], qy[i] < n$.

All squares and queries are numbered in 0-indexing.

The function `solve` returns an array containing the answers to the second type of queries. This array should have a length equal to the number of second-type queries. In C++, this array is returned as a type `vector<int>`, and in Python3, it is returned as a list of integers.

It is guaranteed that the function `solve` will be called exactly once during the program's execution.

Testing

You are provided with template files where you can write your solutions: `tree.cpp` and `tree.py`. Also, in C++, you have been provided with a header file `tree.h` containing the definition of the function `solve`.

For your convenience, graders are provided — files `grader.cpp` and `grader.py`. In these files, reading the input from the standard input, the execution of the function `solve`, and the output of the returned value of the function `solve` to the standard output are implemented. In the testing system, these grader files may differ.

To compile your C++ code written in the file `tree.cpp`, use the command

```
g++ -std=c++20 grader.cpp tree.cpp -o grader
```

After executing this command, an executable file named `grader` or `grader.exe` will be created, depending on your operating system, which can be run to input a test in the specified format.

To run your Python code written in the file `tree.py`, use the command `python3 grader.py`, and then you can input a test in the specified format.

If compilation via commands causes you difficulties, for local testing, you can copy the implementation of input and output from the files `grader` into the files `tree` and run the files `tree.cpp` or `tree.py`. However, before submitting your solution to the testing system, you will need to remove these input and output implementations (in particular, you will need to remove the function `main` if you code in C++).

Input

The grader reads the test in the following format:

The first line specifies two integers n, q ($1 \le n, q \le 10^6$) — the number of squares in the hopscotch and the number of actions by Egor.

The next line specifies n integers $a_0, a_1, \ldots, a_{n-1}$ ($0 \le a_i \le n$) — the initial numbers in the squares.

The following line specifies n integers $p_0, p_1, \dots p_{n-1}$ ($-1 \le p_i < n$), defining the parents of the nodes in the tree. If $p_i = -1$, then node i is the root of the tree; otherwise, node p_i is the parent of node i in the tree. It is guaranteed that this array of ancestors forms a valid rooted tree.

In the next q lines, queries are specified. For any $0 \le i < q$, in the i-th query, depending on the type of query, it is specified in the following format:

$1\ s\ x$ ($0 \le s < n, 0 \le x \le n$) — Egor erases the number in square s and writes the number x there. In the function `solve` arguments, it holds that $qt[i] = 1, qx[i] = s, qy[i] = x$.

$2\ u\ v$ ($0 \le u, v < n$) — Egor asks about the coolness of the jump sequence from u to v. In the function `solve` arguments, it holds that $qt[i] = 2, qx[i] = u, qy[i] = v$.

Output

The grader outputs the results of the function `solve` — the answers to the second type of queries.

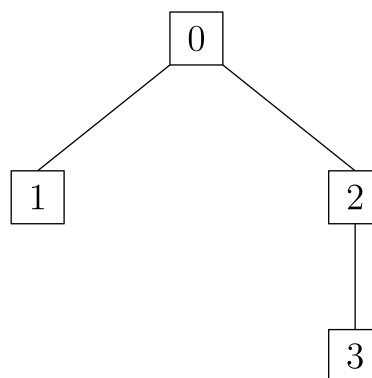
Examples

test	answer
4 3 1 1 2 2 -1 0 0 2 2 0 3 1 2 1 2 0 3	1 2
5 5 0 1 2 3 4 1 2 -1 2 3 2 0 4 1 0 1 1 1 0 2 0 4 2 1 0	5 3 2

Note

In the examples, the input and output data of the grader are specified.

In the first example, the tree of squares looks as follows:



For the first query, consider the jump sequence from 0 to 3:

1. $v_0 = 0, a_0 = 1 \neq 0$, so this square does not add coolness to Egor.
2. $v_1 = 2, a_2 = 2 \neq 1$, so this square also does not add coolness to the jump sequence.

3. $v_2 = 3, a_3 = 2 = 2$, this jump is cool.

Thus, the coolness in the first query is 1.

After the second query, the number in square number 2 is now 1. Therefore, in the second query, jumping to square number 2 becomes cool, and now the answer to this query is 2.

Scoring

The tests for this problem consist of fourteen groups. Points for each group are awarded only if all tests of the group and all tests of some of the previous groups are passed. Note that passing the samples is not required for some groups. **Offline-testing** means that the results of testing your solution on this group will only be available after the competition ends.

Let c_i be the number of squares connected to the i-th square.

Group	Points	Additional constraints		Required groups	Comment
		n	q		
0	0	–	–	–	Samples.
1	10	$n \le 1\,000$	$q \le 1\,000$	0	
2	11	$n \le 200\,000$	$q \le 5\,000$	0, 1	
3	14	$n \le 200\,000$	$q \le 200\,000$	–	No first-type queries
4	5	$n \le 200\,000$	$q \le 200\,000$	–	For exactly one square $c_i = 2$, for the others $c_i \le 3$
5	11	$n \le 200\,000$	$q \le 200\,000$	–	$c_i \le 2$
6	10	$n \le 200\,000$	$q \le 200\,000$		For all second-type queries $v_i = 0$
7	9	$n \le 100\,000$	$q \le 100\,000$	0, 1	
8	8	$n \le 200\,000$	$q \le 200\,000$	0 – 7	
9	17	$n \le 500\,000$	$q \le 500\,000$	0 – 8	
10	1	$n \le 600\,000$	$q \le 600\,000$	0 – 9	**Offline-testing.**
11	1	$n \le 700\,000$	$q \le 700\,000$	0 – 10	**Offline-testing.**
12	1	$n \le 800\,000$	$q \le 800\,000$	0 – 11	**Offline-testing.**
13	1	$n \le 900\,000$	$q \le 900\,000$	0 – 12	**Offline-testing.**
14	1	–	–	0 – 13	**Offline-testing.**