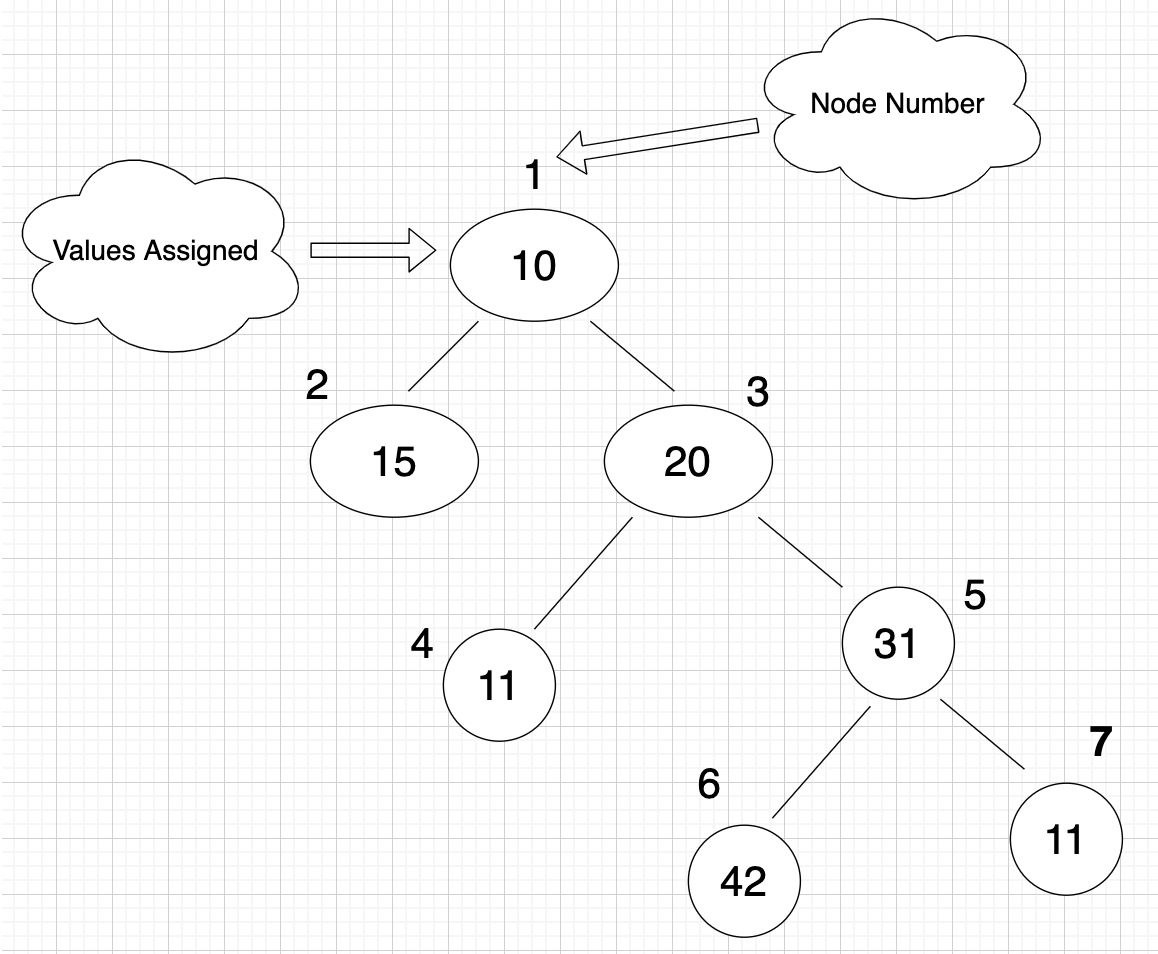
HackTree

You are given a tree with *n* nodes. Each node has a value assigned with it. The cost of a path is defined as the summation of all the values assigned to nodes that belong to the path.

The root of the tree is node number 1.

****Cost of path example****

********

The cost of the path 6 -> 5 -> 3 -> 1 in the above tree is 42 + 31 + 20 + 10 = 103.

A Vertical Path in a tree is the path that is going up towards the root of the tree. It is not necessary for the path to end at the root.

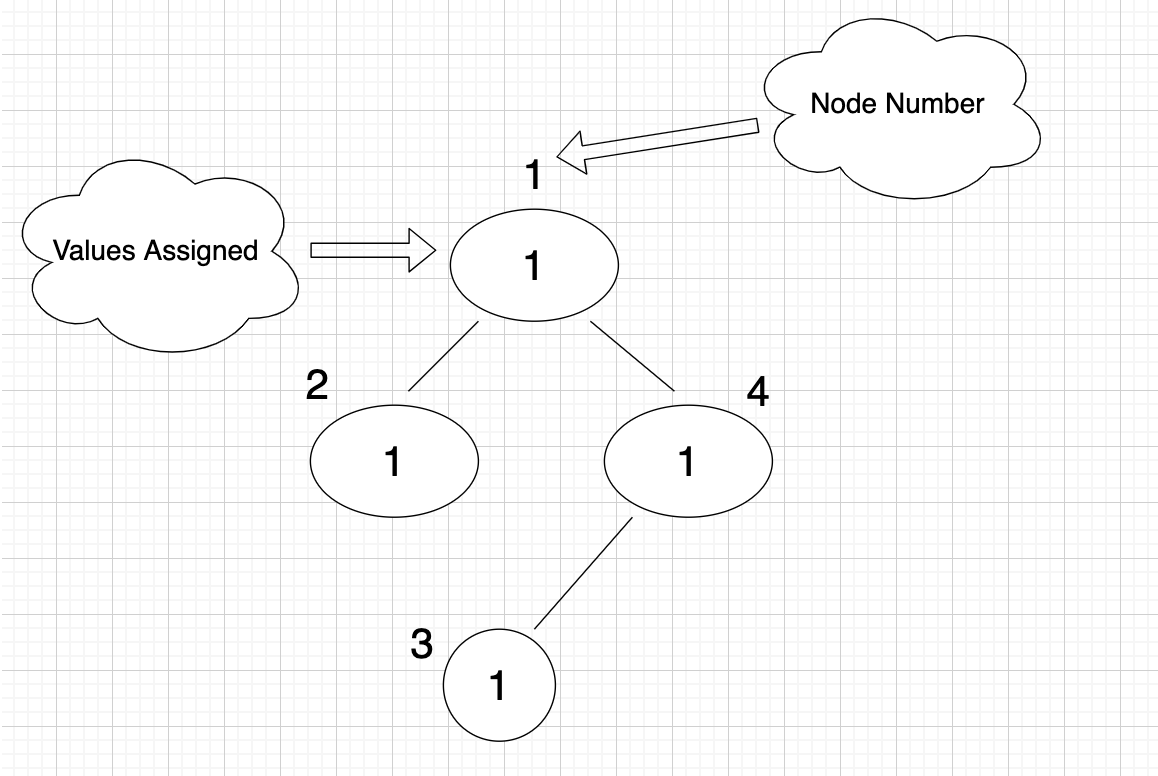
Given a tree with n nodes and an integer *k*. Find the number of vertical pathssuch that the (cost of the path) % *k* = 0 where % represents the modulo operation.

Note: The modulo operation returns the remainder of a division after one number is divided by another. For example - 5 % 2 = 1.

****Example****

*k* = 2

*tree* =



There are a total of 8 vertical paths:

1. 1
2. 2
3. 4
4. 2->1
5. 4->1
6. 3
7. 3->4
8. 3->4->1

But only (2 -> 1), (4 -> 1), (3 -> 4) have (cost of the path) % k = 0.

Hence the answer is 3.

****Function Description****

Complete the function *countVerticalPaths* in the editor below.

*countVerticalPaths* has the following parameters:

*cost*: the array representing the value of each node.

*edge\_nodes*: number of nodes in the tree.

*edge\_from*: integer array where the ith integer denotes one endpoint of the ith edge.

*edge\_to*: integer array where the ith integer denotes the other endpoint of the ith edge

*k*: an integer

****Returns****

*int*: the number of vertical paths with (cost of the path) % *k* = 0

    Note: The tests are generated in such a way that the returned value fits in int32

****Constraints****

* 1 ≤ *n* ≤ 2\*105
* 0 ≤ *cost[i]*  ≤ 108
* 1 ≤ k ≤ 105

Input Format For Custom Testing

The first line contains one integer, *n denoting the size of the cost array*.

The next n lines contain elements of the cost array.

The next line contains two integers, *n and m (*where *m = n - 1)*, denoting the number of nodes and number of roads respectively.

Each line *i* of the *m* subsequent lines (where *0 ≤ i < m*) contains two integers, *u* and *v*, denoting that nodes *u* and *v* are connected via an edge.

The next line contains an integer denoting k.

Sample Case 0

**Sample Input For Custom Testing**

STDIN         FUNCTION

-----         --------

5 -> n = Size of Cost array.

1

2

2

1

2  -> cost = {1,2,2,1,2}

5 4 -> n = 5, m = 4

2 3

2 1

1 4

2 5

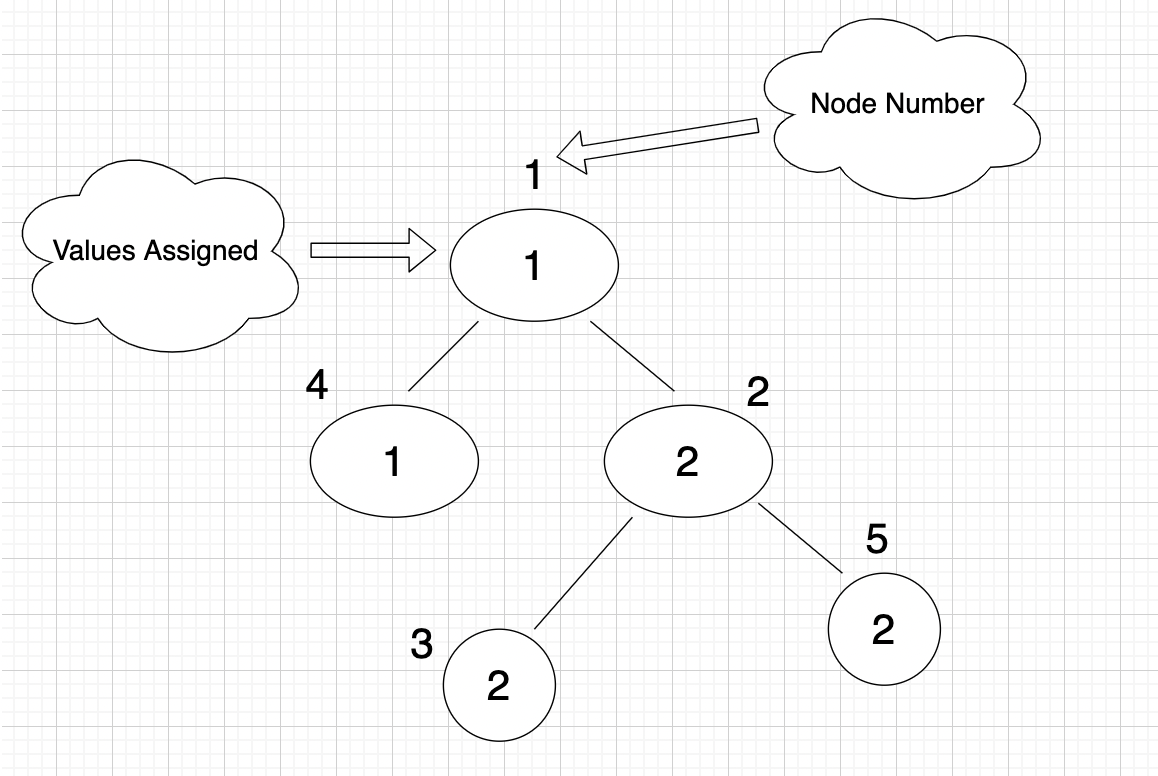
2 -> k = 2

Sample Output

6

**Explanation**

There are 6 vertical paths following the given condition:



these paths are:

1. 2

2. 5 -> 2

3. 4 -> 1

4. 3

5. 3 -> 2

6. 5

Hence the answer is 6.

Sample Case 1

**Sample Input For Custom Testing**

STDIN         FUNCTION

-----         --------

5 -> n = 5

2

3

0

3

0  -> cost = {2,3,0,3,0}

5 4 -> n = 5, m = 4

2 3

3 1

3 4

3 5

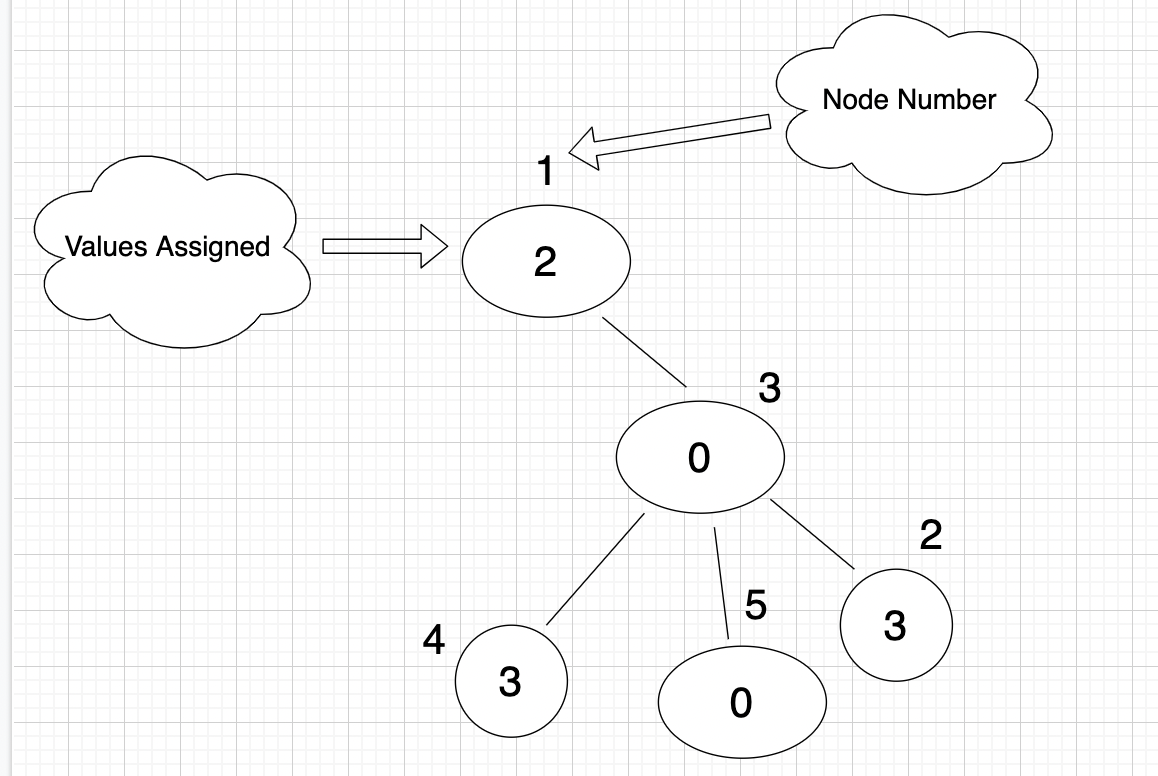
3 -> k = 3

**Sample Output**

7

**Explanation**

There are 7 vertical paths following the given condition :



these paths are :

1. 4

2. 4 -> 3

3. 2

4. 2 -> 3

5. 5

6. 3

7. 5 -> 3

Hence the answer is 7.

import java.io.\*;

import java.math.\*;

import java.security.\*;

import java.text.\*;

import java.util.\*;

import java.util.concurrent.\*;

import java.util.function.\*;

import java.util.regex.\*;

import java.util.stream.\*;

import static java.util.stream.Collectors.joining;

import static java.util.stream.Collectors.toList;

class Result {

/\*

\* Complete the 'countVerticalPaths' function below.

\*

\* The function is expected to return an INTEGER.

\* The function accepts following parameters:

\* 1. INTEGER\_ARRAY cost

\* 2. UNWEIGHTED\_INTEGER\_GRAPH edge

\* 3. INTEGER k

\*/

/\*

\* For the unweighted graph, <name>:

\*

\* 1. The number of nodes is <name>Nodes.

\* 2. The number of edges is <name>Edges.

\* 3. An edge exists between <name>From[i] and <name>To[i].

\*

\*/

public static int countVerticalPaths(List<Integer> cost, int edgeNodes, List<Integer> edgeFrom, List<Integer> edgeTo, int k) {

}

}

public class Solution {

public static void main(String[] args) throws IOException {

BufferedReader bufferedReader = new BufferedReader(new InputStreamReader(System.in));

BufferedWriter bufferedWriter = new BufferedWriter(new FileWriter(System.getenv("OUTPUT\_PATH")));

int costCount = Integer.parseInt(bufferedReader.readLine().trim());

List<Integer> cost = IntStream.range(0, costCount).mapToObj(i -> {

try {

return bufferedReader.readLine().replaceAll("\\s+$", "");

} catch (IOException ex) {

throw new RuntimeException(ex);

}

})

.map(String::trim)

.map(Integer::parseInt)

.collect(toList());

String[] edgeNodesEdges = bufferedReader.readLine().replaceAll("\\s+$", "").split(" ");

int edgeNodes = Integer.parseInt(edgeNodesEdges[0]);

int edgeEdges = Integer.parseInt(edgeNodesEdges[1]);

List<Integer> edgeFrom = new ArrayList<>();

List<Integer> edgeTo = new ArrayList<>();

IntStream.range(0, edgeEdges).forEach(i -> {

try {

String[] edgeFromTo = bufferedReader.readLine().replaceAll("\\s+$", "").split(" ");

edgeFrom.add(Integer.parseInt(edgeFromTo[0]));

edgeTo.add(Integer.parseInt(edgeFromTo[1]));

} catch (IOException ex) {

throw new RuntimeException(ex);

}

});

int k = Integer.parseInt(bufferedReader.readLine().trim());

int result = Result.countVerticalPaths(cost, edgeNodes, edgeFrom, edgeTo, k);

bufferedWriter.write(String.valueOf(result));

bufferedWriter.newLine();

bufferedReader.close();

bufferedWriter.close();

}

}

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*