Ball Machine

Madina has invented a ball machine to entertain the participants of the IOI. The internal structure of the machine is as follows:

- The machine consists of N **nodes**, indexed from 0 to $N-1$. Node $N-1$ is called the **root**.
- For each node u with $0 \leq u < N-1$, the **parent** of node u is a node $P[u]$ with $P[u] > u$, and node u is a **child** of $P[u]$. The root has no parent. A node with no children is called a **leaf**.

You do **not** know N or the parent $P[u]$ of any node u. Instead, Madina tells you M, the number of leaves, and that the indices of the leaves are $0, 1, \ldots, M-1$. Your task is to determine the internal structure of the machine by operating it.

Each node of the machine can hold at most one **ball**, and each ball has a **value** which is a non-negative integer. We say that a node has value x if it holds a ball with value x. A node is **occupied** if it contains a ball; otherwise, it is **free**. Initially, every node is free.

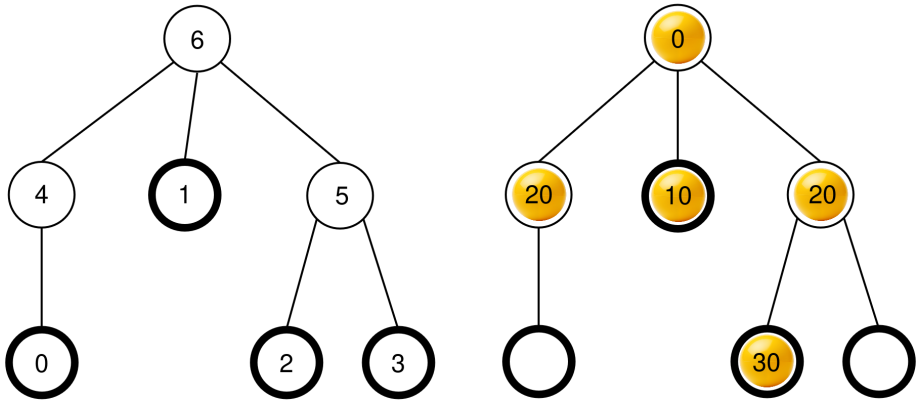
You can perform two kinds of operations:

- **insert**(U, X): attempt to insert a new ball with value X at leaf U.
  - If leaf U is occupied, the operation returns `false` without inserting the ball.
  - If leaf U is free, the ball is placed at node U. Then the ball repeatedly moves to the parent of its current node as long as that parent is free. The movement stops when the ball reaches the root or a node whose parent is occupied. The operation returns `true`.
- **collect()**: if the root is free (which means the machine is empty), `collect` returns an empty array. Otherwise, the recursive procedure `traverse` described below gathers the values of all balls that are currently in the machine into an array S. Initially, the array S is empty and `traverse(N - 1)` is called.

```
traverse(u):
  append the value of the ball at node u to the end of S
  let c[u] be the list of the occupied children of u
  sort c[u] in non-decreasing order according to the values
    of the balls at those nodes (if multiple nodes have the
    same value, they can appear in any order)
  for each v in c[u]:
    traverse(v)
```

After this procedure completes, **all balls are removed** from the machine, and `collect` returns S.

For example, consider a machine with $N = 7$ nodes and parent array $P = [4, 6, 5, 5, 6, 6]$. The picture on the left displays the empty machine with node indices, while the picture on the right shows a possible configuration of balls after some sequence of `insert` operations (this sequence is given in the Example section).



When `collect()` is called, the root (node 6) is occupied, so S is initialized as $S = [\,]$ and `traverse(6)` is called.

`traverse(6)`: node 6 has value 0; appending it to S gives $S = [0]$. The children of node 6 are 4, 1, and 5, all of which are occupied. Their values are 20, 10, and 20, respectively. Sorting in non-decreasing order yields $c[6] = [1, 5, 4]$ (note that $c[6] = [1, 4, 5]$ would also be valid, since nodes 4 and 5 have the same value). The procedure then calls `traverse` on each node in $c[6]$ in that order.

- **`traverse(1)`**: node 1 has value 10; appending it to S gives $S = [0, 10]$. Node 1 has no occupied children, so this call ends.
- **`traverse(5)`**: node 5 has value 20; appending it to S gives $S = [0, 10, 20]$. Node 5 has one occupied child, node 2, so $c[5] = [2]$ and the procedure calls `traverse(2)`.
    - **`traverse(2)`**: node 2 has value 30; appending it to S gives $S = [0, 10, 20, 30]$. Node 2 has no occupied children, so this call ends.
    - The execution returns to `traverse(5)` which has processed all children in $c[5]$, so its call ends.
- **`traverse(4)`**: node 4 has value 20; appending it to S gives $S = [0, 10, 20, 30, 20]$. Node 4 has no occupied children, so this call ends.

All calls have completed and there are no more nodes to process. Finally, every ball is removed from the machine and `collect` returns the array $S = [0, 10, 20, 30, 20]$.

Your task is to determine the number of nodes N and report an array of parents $R = [R[0], R[1], \ldots, R[N-2]]$ that describes the structure of the machine, but with a potentially different labeling of the nodes that are not leaves and not the root. Formally, a reported array R is considered correct if it is possible to assign a distinct label $L[u]$ with $0 \leq L[u] < N$ to each node u of the machine such that:

- $L[u] = u$ for all $0 \leq u < M$ and $u = N - 1$, and
- $L[P[u]] = R[L[u]]$ for all $0 \leq u < N - 1$.

It is **not** required that $R[u] > u$. The machine **must be empty** when you report your solution.

Let K be the number of `collect` operations performed, and B be the maximum value of any ball across all `insert` calls. Let $C = K + B$. Then C **must not exceed** 1000**.** Your score in some subtasks depends on the value of C.

Implementation Details

You should implement the following procedure.

```
std::vector<int> find_structure(int M)
```

- M: the number of leaves in the machine.
- The procedure is called exactly once for each test case.
- The procedure must return an array $R = [R[0], R[1], \ldots, R[N-2]]$ of length $N - 1$, a correct array of parents.

To interact with the machine, your procedure can call the following two procedures.

The first procedure is:

```
bool insert(int U, int X)
```

- U: the index of the leaf where the ball should be inserted. The condition $0 \leq U < M$ must be met.
- X: the integer value of the ball. The condition $0 \leq X \leq 1000$ must be met.
- This procedure returns `true` if the ball was successfully inserted, or `false` if leaf U was already occupied.
- This procedure can be called at most $500\,000$ times.

The second procedure is:

```
std::vector<int> collect()
```

- Collects the values on all inserted balls, empties the machine, and returns the collected array S.

If at any point during the execution of your program the value of C exceeds 1000, your solution will receive the verdict `Output isn't correct: Too many resources used`.

The structure of the machine is fixed before `find_structure` is called. The grader is **deterministic** in the sense that if you run it twice and both runs perform the same sequence of operations, the calls to `collect` will return the same arrays.

Constraints

- $2 \leq N \leq 1000$
- $1 \leq M \leq 200$
- $M < N$

Subtasks

Subtask	Score	Additional Constraints
1	5	The root has exactly M children.
2	10	$M \leq 3$
3	25	$N \leq 200$, $M \leq 45$
4	60	No additional constraints.

In subtasks 3 and 4, your score depends on the value of C as follows.

Subtask 3 (25 points)

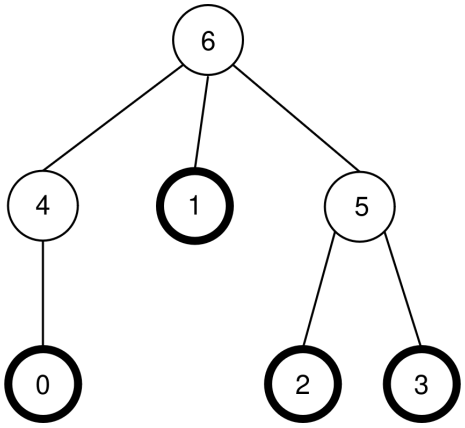
Limits	Score
$1000 < C$	0
$45 < C \leq 1000$	13
$C \leq 45$	25

Subtask 4 (60 points)

Limits	Score
$1000 < C$	0
$200 < C \leq 1000$	7
$71 < C \leq 200$	$47 - \frac{C}{5}$
$44 < C \leq 71$	$104 - C$
$C \leq 44$	60

Example

Consider the same machine as in the task description, so $N = 7$, $M = 4$, and $P = [4, 6, 5, 5, 6, 6]$. The machine is again shown in the following figure:



The grader makes the following call:

```
find_structure(4)
```

The procedure can make the following sequence of calls:

1. **`insert(0, 0)`**: leaf 0 is free, so a ball with value 0 is placed at leaf 0. Node $P[0] = 4$ is free, so the ball moves to node 4. Node $P[4] = 6$ is free, so the ball moves to node 6. Since node 6 is the root, the movement stops. The call returns `true`.
2. **`insert(3, 20)`**: leaf 3 is free, so a ball with value 20 is placed at leaf 3. Node $P[3] = 5$ is free, so the ball moves to node 5. Since $P[5] = 6$ is occupied, the movement stops. The call returns `true`.
3. **`insert(1, 10)`**: leaf 1 is free, so a ball with value 10 is placed at leaf 1. Since $P[1] = 6$ is occupied, the ball stays at node 1. The call returns `true`.
4. **`insert(2, 30)`**: leaf 2 is free, so a ball with value 30 is placed at leaf 2. Since $P[2] = 5$ is occupied, the ball stays at node 2. The call returns `true`.

5. **`insert(1, 25)`**: leaf 1 is occupied, so the ball is not placed at leaf 1. The call return `false`.
6. **`insert(0, 20)`**: leaf 0 is free, so a ball with value 20 is placed at leaf 0. Node $P[0] = 4$ is free, so the ball moves to node 4. Since $P[4] = 6$ is occupied, the movement stops. The call returns `true`.

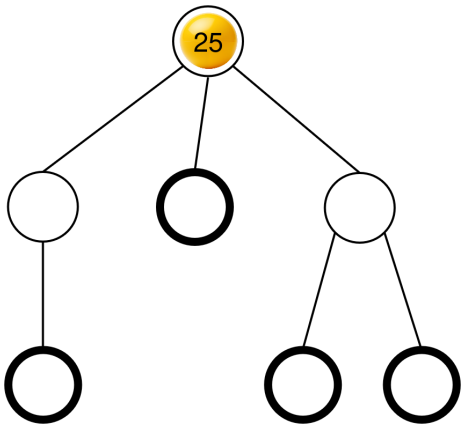
The resulting configuration of balls was already displayed in the task description.

Next, the procedure can call:

```
collect()
```

The execution of this call was already explained in the task description, so this call returns $S = [0, 10, 20, 30, 20]$. Afterwards, the machine is once again empty.

Next, the procedure can call **`insert(2,25)`**. Leaf 2 is free, so a ball with value 25 is placed at leaf 2. This ball then moves to node 5, and then to node 6, where it stops. The call returns `true`. The resulting configuration of balls is shown in the following figure.



Finally, the procedure can call **`collect()`** which returns $S = [25]$ and empties the machine.

There are two correct parent arrays that `find_structure` can return: $R = P = [4, 6, 5, 5, 6, 6]$ (which corresponds to the labeling $L = [0, 1, 2, 3, 4, 5, 6]$), and $R = [5, 6, 4, 4, 6, 6]$ (which corresponds to the labeling $L = [0, 1, 2, 3, 5, 4, 6]$). In this example, $K = 2$ and $B = 30$, so $C = 32$.

Sample Grader

Input format:

```
N M
P[0] P[1] P[2] ... P[N-2]
```

Output format:

```
K B C
Q
R[0] R[1] R[2] ... R[Q-1]
```

Here, Q is the length of the array R returned by `find_structure`.