

Reconstruct

Time limit: 4 seconds

Memory limit: 1024 MiB

This is an interactive problem.

Bissy is about to enter the volunteer organised orienteering game *Final destination*, where EJOI delegations will compete to visit different locations across the city. But Bissy is more interested in *how* the game was designed than winning it.

There are N locations on the map and N teams. Every team has to visit all locations and have unique starting locations — team i starts from location i . Each team is given their own route.

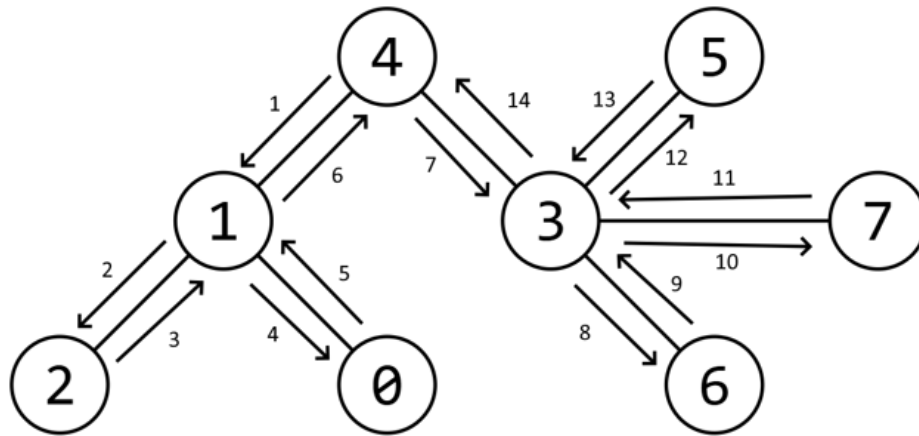
Bissy also has some insider information — the routes are based on a hidden structure of fast public transport lines. More precisely, the jury of the competition chose $N - 1$ lines, each connecting a pair of locations, such that every location is reachable from any other using these lines. Such a structure is also called a tree. Then, for each starting location i , the jury generated an **arbitrary DFS walk** (defined below) and assigned this walk as the route for team i .

Bissy wants to figure out the public transport tree that the jury selected. She only asks quick questions of the kind:

- What is the j -th destination of the i -th team?

Help Bissy by writing a program `find_tree` that finds the hidden tree.

A *DFS walk* of a tree is the order in which its vertices are visited for the first time by a depth-first-search procedure. Such a procedure starts at a given vertex v and recursively goes to one of its unvisited neighbours until no such neighbour exists — then it goes back to the previously visited vertex and continues from there. Example:



Here $v = 4$ and the arrows along the edges correspond to the steps of a depth-first-search. The walk generated is $[4, 1, 2, 0, 3, 6, 7, 5]$. Note that the order in which we visit neighbours matters; another walk could be $[4, 3, 5, 7, 6, 1, 0, 2]$.

The tree structure and N DFS walks are fixed before your program starts and does not change in response to your questions. Orders of neighbours of the vertices used during such DFSs can be different.

Implementation details

You have to implement the following function:

```
std::vector<std::pair<int, int>> find_tree(int N)
```

- here N is the number of locations;
- the return value is a list of length $N - 1$ of the edges of the tree — the order of the edges and their endpoints does not matter.

For each test, this function can be called up to T times.

To interact with the jury, you can make calls to the following function:

```
int guess(int i, int j)
```

It returns the j -th location in the i -th walk of the tree. Note that `guess(i, 0)` will return i . You can assume that the function responds in constant time $O(1)$ for all subtasks up to 6, and in logarithmic time $O(\log N)$ for subtask 7. It should hold that $0 \leq i, j \leq N - 1$, otherwise your solution will receive the verdict `Output isn't correct: Invalid call`.

Constraints

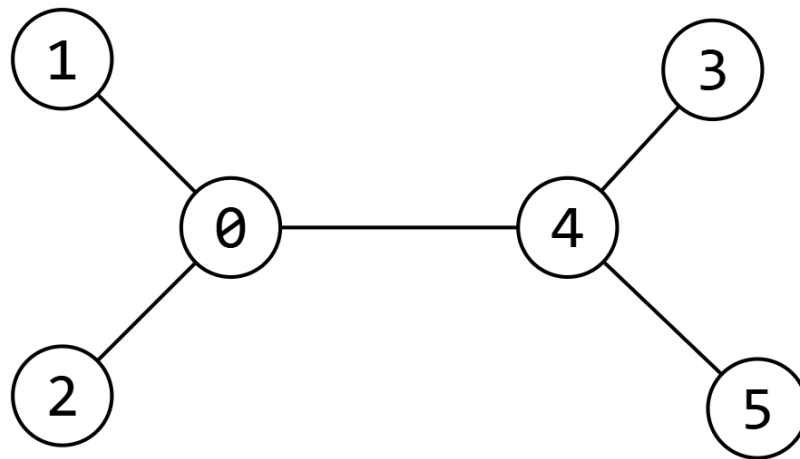
- $2 \leq N \leq 2^{16} + 1$

- T follows the constraints based on the maximum N (denoted by N_{max}) among the function calls in one test:
 - if $N_{max} \leq 9$, then $1 \leq T \leq 100$
 - if $N_{max} \leq 2^{10} + 1$, then $1 \leq T \leq 10$
 - if $N_{max} \leq 2^{16} + 1$, then $1 \leq T \leq 3$

The system grader can use up to 280 MiB, which counts toward your solution's memory.

Example

Assume the hidden structure of fast public transport lines is as follows:



Suppose that for the starting location 0, the walk is $[0, 1, 2, 4, 3, 5]$. Then the following is an example interaction:

Participant program	Jury program
	<code>find_tree(6)</code>
<code>guess(0, 0)</code>	0
<code>guess(0, 1)</code>	1
<code>guess(0, 2)</code>	2
<code>guess(0, 3)</code>	4
<code>guess(0, 4)</code>	3
<code>guess(0, 5)</code>	5
<code>return {{0,1},{0,2},{4,0},{5,4},{3,4}};</code>	

Note that the information from the queries is not enough to uniquely determine the tree, but this is in fact the answer to the test in subtask 0.

Subtasks

Subtask	Points	N	Additional constraints
0	0	-	The example.
1	11	≤ 9	-
2	6	≤ 100	Every vertex is connected to at most 2 others.
3	13	≤ 100	Every walk is generated by a depth-first-search which at every step prioritizes moving away from vertex 0. If there are multiple options to move away from vertex 0, one of them is chosen arbitrarily.
4	11	≤ 100	Any vertex other than 0 is connected to at most 2 others.
5	10	≤ 100	-
6	31	$\leq 2^{10} + 1$	-
7	18	$\leq 2^{16} + 1$	-

Scoring

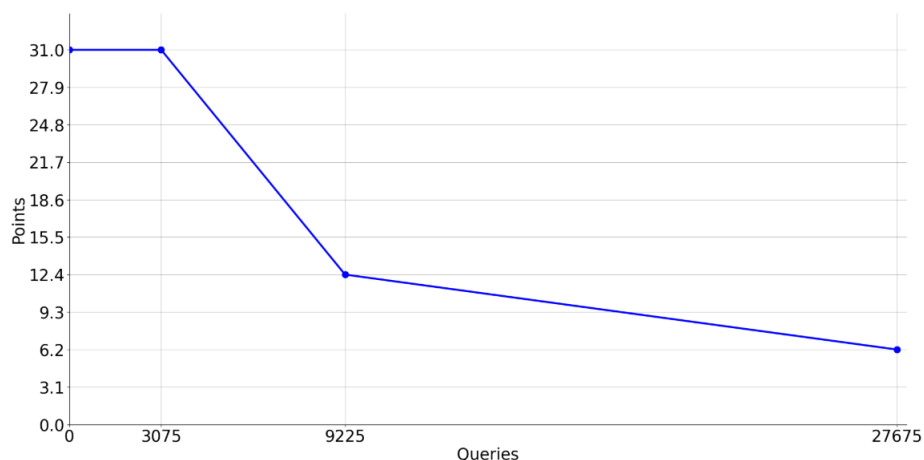
For each subtask up to subtask 5, you gain full points if you successfully find the tree within the time limit constraints. For subtasks 6 and 7, the fraction of the full points S for a given test depends on the **maximum** number of queries you ask on one of its subtests, Q_{max} . The scoring for these subtasks is as follows:

- if $Q_{max} \leq L_1$, then $S = 1.0$;
- if $L_1 < Q_{max} \leq L_2$, then $S = 0.4 + 0.6 \times \frac{L_2 - Q_{max}}{L_2 - L_1}$;
- if $L_2 < Q_{max} \leq 3L_2$, then $S = 0.2 + 0.2 \times \frac{3L_2 - Q_{max}}{2L_2}$;
- if $3L_2 < Q_{max}$, then $S = 0.2$;

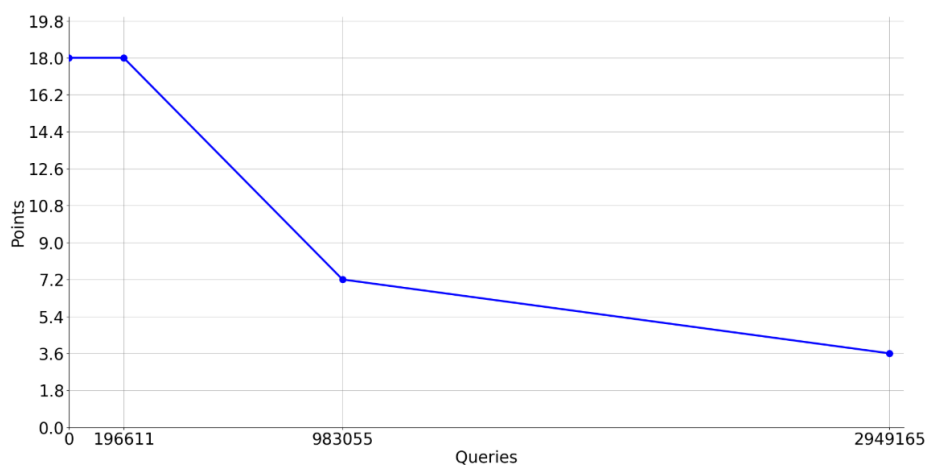
where $L_1 = 3 \times (2^{10} + 1) = 3075$ and $L_2 = 9 \times (2^{10} + 1) = 9225$ for subtask 6, and $L_1 = 3 \times (2^{16} + 1) = 196611$ and $L_2 = 15 \times (2^{16} + 1) = 983055$ for subtask 7. The fraction of points you get from the whole subtask is the minimum S among all tests.

The graphs below show the scoring function for the last two subtasks:

Subtask 6



Subtask 7



Sample grader

There are two sample graders provided.

For local testing: `Lgrader.cpp` can be compiled together with your program to provide a tool for testing. The program reads the number T of test cases and for each one expects the number of location N followed by $N - 1$ lines, each with a pair of integers describing a direct line. After that N lines of N integers follow – the DFS walks. Note that walk i has to start from vertex i . You can let the grader generate walks for you by setting the `AUTO_GENERATE` constant to `true`. As an output the grader either reports an error message if the result is incorrect or the number of queries asked for each test case and an overall maximum. Also note that the grader does not work for sufficiently large N (more precisely subtask 7 constraints), you are free to use your imagination for that case.

For system user tests: `stub.cpp` can be used together with your program for the user tests on the system. The input format is the same as the one for the other grader. Note that for user tests there isn't a built-in option for walk auto generation.