Alexandru Velea
University of Bucharest

Basic observations:

- We can consider an undirected graph where the nodes are cities and the edges are the roads between cities. From the fact that a road will not be built between two cities that are connected by a path in the graph, we can deduce that our graph is a union of trees.
- Creating a new road is equivalent to joining two trees.
- In the beginning, there are no edges in the graph, hence, the graph contains `N` trees with size `1`.

Solution 1 – 7 points

In order to find a newly created road, we can just query each pair of nodes `(i, j)` (except for those that we know are already joined by a road), by placing i in the first set and j in the second. If the query returns `true`, those 2 nodes are connected by the newly created road. If we want to find the T^{th} road, (meaning that we know `T - 1` roads), we will perform `N * (N - 1) / 2 - (T - 1)` queries. In total, we will make $O(N^3)$ queries.

Solution 2 – 18 points

Every time we want to find a new road, for each node `i`, we will make a query of the following type: the first set will contain only node `i`, and the second set should only contain the other nodes which don't share an edge with `i`.

For example, if `N = 6` and we know three edges: `(1, 2), (2, 3), (1, 4)`, for node `1`, the first set will be `{1}` and the second set should contain `{3, 5, 6}`. It's not mandatory to exclude `3` from the second set, even though it's impossible to have an edge between 1 and 3.

Using this approach, if the query returns `true` for a node `A`, we know for sure that `A` is one of the two nodes united by the current edge. By doing this for each node, we'll find the two nodes for which the query returned `true`, hence finding the nodes which are connected by the new edge.

This solution requires $O(N^2)$ queries.

Further observations

To describe the other solutions more easily, let's define some additional notations and point some observations.

At time `T`, when we need to guess the T^{th} road, we have `N - T + 1` connected components figured out. We number this components with numbers from `0` to `N - T`, in arbitrary order.

When we say that a connected component is included in one of the two sets of a query, we mean that all its nodes are included in that set.

Solution 3 – 61 points

When finding a road, suppose we were able to assign all connected components to two sets such that the newly created edge is between two components that are not in the same set.

We can now consider the set `A` comprised of all the nodes from the first set of connected components, and set `B` containing all the nodes from the second set of connected components. To find the edge, we will find the two nodes that it connects independently.

To find the first node, we will always perform queries where the second set of the query is `B`. The first set of a query will represent candidate nodes from `A` (nodes that may still connected to the new edge). We will reduce `A` to a single node (and hence find the correct one) the following way: split `A` in half and make a query for this subset. If the query returns `true`, we keep this half of `A`, else we take the other one.

The same process can be repeated to find the other node from set `B`. The new edge is given by this pair.

Now, how do we find the two sets of connected components in the first place? One solution to find them is to get the array of the `N - T` connected components, random shuffle it and take the first half as one of the sets and the other half as the other set. We can then make a query with sets `A` and `B` to ensure that the edge is indeed between nodes of connected components that belong to different sets.

Because the grader is built in order to maximize the number of queries, this approach takes approximately `2 * log(N)` steps, if we have `N` connected components at a given time. All in all, this approach makes approximately `N * 4 * log(N)` queries in total.

Solution 4 – 90 points

Another approach to split the connected components into two sets is to label them according to the bits that differ between the binary representations of the components' indices. We try this type of separation for all the `log N` bits. For example, if we separate them using the least significant bit, the first set will contain all the components with odd indices and the other set will contain all the components with even indices.

Repeat this at most `log N` times until we find a correct way to separate them. This will always happen, because if all the bits are the same for the two target components, then it means that the indices of the two connected components that will be joined by the edge must be equal, and this cannot happen under the constraints of the problem.

After that just use the same approach as in the third solution to find the two nodes that were connected by the newest road.

The number of queries for this solution in the worst case is `sum(ceil(log2(i)) + ceil(log2(i / 2)) + ceil(log2(n - (i / 2))))` for `i` from `2` to `N`, which is around 1705 if the grader tries to maximise the number of queries needed (by answering `false` to as many queries as possible). In total, this approach makes approximately `N * 3 * log N` queries.

Solution 5 – 100 points

Instead of assigning the connected components to two sets and then finding the target nodes from those sets, it's better to first find the two components that are joined by the new edge and find the target nodes from those components only.

Let's look at the binary representation of the two components that we're looking for. Let `A` and `B` be the indices of those components.

First of all, we need to compute `A xor B`.

To do this we perform `log M` queries where `M` is the current number of connected components. The two query sets for the i^{th} bit contain all components, the first set containing the ones which have the i^{th} bit equal to `0` and the second one the i^{th} bit equal to `1`. If the query returns `true`, we know that the i^{th} bit of `A xor B` is `1`; otherwise it is `0`.

After doing all `log M` queries, we select one bit (let's call it `X`) which is equal to `1` and say that `A` has that bit set to `0` and `B` has it equal to `1`. In order to determine the other bits, we do the following process.

If the i^{th} bit is the same in `A` and `B`, we split the components into two sets: the first one should have all the components which have the i^{th} bit `0` and the X^{th} bit equal to `0`, and the second set should contain all the components which have i^{th} bit equal to `0` and the X^{th} bit equal to `1`. If the query returns `true`, we know that the i^{th} bit is `0`, otherwise it is `1`, for both `A` and `B`.

If the i^{th} bit is different between `A` and `B`, we split the components into two sets: the first one should have all the components which have the i^{th} bit `0` and the X^{th} bit equal to `0`, and the second set should contain all the components which have i^{th} bit equal to `1` and the X^{th} bit equal to `1`. If the query returns `true`, we know that `A` has that bit equal to `0` and `B` has that bit `1`. If not, `A` has that bit `1` and `B` has the i^{th} bit equal to `0`.

After determining the two components, apply the same procedure as before to find the two target nodes.

The total number of queries can be determined by the formula:

```
Sum for i from 2 to N of:
        ceil(log2(i))        – determining A xor B
      + ceil(log2(i) - 1     – determining A and B
      + ceil(log2(i-1))      – worst case for querying the two components to find the edge
```

In practice, the maximum number of queries performed by this approach is `1613`.