Game Solution

Let E_{yes} be the set of edges about which the contestant has answered "yes" (connected), E_{no} the set of edges about which contestant has answered "no", and E_{maybe} the rest of the edges, whose statuses are not yet determined. Also, let $G = (V, E_{\text{yes}})$ and $H = (V, E_{\text{yes}} \cup E_{\text{maybe}})$. G is the graph you get by assuming that every edge in E_{maybe} are not connected, while H is the graph you get by assuming that all edges in E_{maybe} are connected.

Initially, G is empty and thus not connected, while H is connected. In order not to reveal any clue to the judge, the contestant should maintain the invariant: G should always be disconnected, while H should always be connected.

There are several possible ways to maintain the invariant.

An $O(n^4)$ solution

When asked by the judge whether an edge $e = (u, v)$ is connected, answer "no" if and only if e is part of a cycle in H. One can see that this does not change the connectivity of G and H.

To decide whether e forms a circle, one can perform a depth-first search to find out whether there is a path from u to v in $(V, E_{\text{yes}} \cup E_{\text{maybe}} - (u, v))$. This is an $O(n^2)$ operation. As there are $O(n^2)$ edges, the total running time is $O(n^4)$.

In other words, we answer "yes" if and only if e is a bridge in H.

An $O(n^2)$ solution

Given a vertex v, let $D(v)$ be the connected component v belongs to in G. We maintain two data structures:

1. R is a table mapping each v to a representative of $D(v)$.
2. S is a symmetric matrix indexed by V. For u and v in V, if $R(u) \neq R(v)$, $S(R(u), R(v))$ is the number of edges, in E_{maybe}, that connects $D(u)$ and $D(v)$.

The contestant answers "yes" to query (u, v) if and only if $S(R(u), R(v)) = 1$.

R can be implemented as a disjoint-set linked list. Each disjoint set is represented by a linked list of its elements, and the representative is the one at the head. Each element has a pointer to its representative. To unite two sets we connect the lists, and update the pointers. An union takes $O(n)$ time and a find takes $O(1)$ time.

As for S, initially $S(u, v) = 1$ unless $u = v$. Whenever the judge asks about (u, v), S is updated as follows.

1. If the contestant answers "no", we decrement $S(R(u), R(v))$ by 1.
2. If the contestant answers "yes", let w be the representative after uniting $D(u)$ and $D(v)$. For each x that is a representative of some connected component, both $S(w, x)$ and $S(x, w)$ are updated to $S(R(u), x) + S(R(v), x)$.

There can be at most $n - 1$ unions, thus the total time spent on union is $O(n^2)$. An update of S requires $O(1)$ time for a "no" response, and $O(n)$ time for a "yes" response. Since the graph G is a tree, we respond "yes" exactly $n - 1$ times. Thus the time spent on updating S is also $O(n^2)$. We thus have an $O(n^2)$ algorithm.

An One-Liner $O(n^2)$ Algorithm

There is a surprising one-line $O(n^2)$ algorithm:

```
#include "game.h"

void initialize(int n) {
    // DO NOTHING!
}

int c[1500];
int hasEdge(int u, int v) {
    return ++c[u > v ? u : v] == (u > v ? u : v);
}
```

To understand the algorithm, imagine that we partition the set of all the possible edges into $E_1, E_2, \ldots E_{n-1}$, with $E_i = \{(i, j) \mid i > j\}$. Each E_i has exactly i possible edges. The algorithm above answers "yes" to (u, v) (where $u > v$) if it is the last edge in E_u that is queried.

To see how it works, consider the last query. Denote the queried edge by e, and the graph $G = (V, E_{\text{yes}} - e)$. The contestant wins if G is disconnected, while $G + e$ is connected.

- G is disconnected, since it contains only $n - 2$ edges.
- $G + e$ is connected, since it contains $n - 1$ edges, and there is no cycle in $G + e$. One can see that there is no cycle since, in each E_i, we answer yes to only one edge. Formally, if there is a cycle C in $G + e$, considering the node u in C with largest id, E_u must has exactly one edge in $G + e$. But u has two neighbors in C with smaller ids, a contradiction.