As usual this statement does not assert that the problem can be solved with only \( n \) comparisons, only that \( n \) are necessary.

In fact, the best known algorithm does \( n - 1 \) comparisons, much worse than \( n \).

\[
\text{FindMax(A)}
\]

1. \( n \leftarrow \text{length}(A) \)
2. \( \text{max} \leftarrow A[1], \text{imax} \leftarrow 1 \)
3. \( \text{for} \ i \leftarrow 2 \ \text{to} \ n \)
4. \( \text{if} \ A[i] > \text{max} \)
5. \( \text{max} \leftarrow A[i] \)
6. \( \text{imax} \leftarrow i \)
7. \( \text{return} (\text{max}, \text{imax}) \)

\[
\text{Ex} \quad n = 4
\]

\[
\text{Decision Tree}: \quad \# \text{Comp} = \log_2 4 = 2
\]

\[
\text{Best known :} \quad \# \text{Comp} = 3
\]

**Exercise**

Draw a decision tree for the operation of \( \text{FindMax} \) in this case. Observe that its height is 3 not 2.
Ex \( n = 100 \)

Deci\sion Tree: \# Comp = \( \log_2 100 \) = 10
Best Known: \# Comp = 1000

We must either find a better algorithm (not possible) or obtain a tighter lower bound.

Adversary Argument:

Consider any comparison based algorithm for this problem and let it run on an array \( A[1..n] \), as yet unspecified.

Daemon's strategy:
Answer each question concerning a comparison as if \( A[i] = i \) (1 ≤ i ≤ n), i.e., as if \( A = (1, 2, \ldots, n) \). In other words, when the algorithm asks "Is \( A[i] < A[j] \)?",

the daemon answers

\[
\begin{cases} \text{true} & \text{if } i < j \quad ("i \text{ has lost a comparison}") \\ \text{false} & \text{if } j < i \quad ("j \text{ has lost a comparison}") \end{cases}
\]

When this happens we say that the smaller of \( i \) and \( j \) has "lost a comparison".
Now assume that the algorithm does fewer than \( M = n - 1 \) comparisons before it halts and outputs the index \( k \) (on the pair \((A[k], k)\)), i.e., the algorithm claims that \( A[k] \) is maximum in array \( A \).

Let \( j \) be an integer satisfying

- \( 1 \leq j \leq n \)
- \( j \neq k \)
- \( j \) has not lost any comparisons.

Such an integer must exist since, by assumption, at most \( n - 2 \) comparisons have been performed, and each new comparison creates at most one new loser, hence there are at most \( n - 2 \) losers.

At this point, the algorithm can say the algorithm is wrong by claiming that array \( A \) is given by

\[
A[i] = \begin{cases} 
  i & i \neq j \\
  n+1 & i = j 
\end{cases}
\]

indeed \( A[k] = k \) is not maximum in
This array, yet the demons answers are all consistent with it.

Therefor no correct algorithm can solve this problem with fewer than \( m=n-1 \) comparisons, and our best known algorithm cannot be improved upon.

**Graph Connectivity**

Let \( G=(V,E) \) be an (undirected) graph on \( |V|=n \geq 2 \) vertices.

Problem: Determine whether or not \( G \) is connected.

We consider only algorithms which are allowed to ask questions of the form "is vertex \( u \) adjacent to vertex \( v \)?"

The decision tree lower bound is trivial:

\[
\text{#outcomes per question} = 2 \quad (\text{yes/no})
\]

\[
\text{#verdicts} = 2 \quad (\text{connected/disconnected})
\]

\[
h \geq \lceil \log_2 2 \rceil = 1
\]

\[
\text{At least 1 question is necessary.}
\]
Depth first search (DFS) solves this problem in time $\Omega(n^2)$. (See section 22.3 for a description.)

**Adversary lower bound:**

Consider any "adjacency" based algorithm for this problem and start it on an (unspecified) graph $G = (V, E)$ with $n = |V|$. 

*Daemon's strategy:*

Partition $V$ into two subsets $X$ and $Y$ of sizes $\lfloor n/2 \rfloor$ and $\lceil n/2 \rceil$ respectively, i.e.

$$X \cup Y = V, \quad X \cap Y = \emptyset, \quad |X| = \lfloor n/2 \rfloor, \quad |Y| = \lceil n/2 \rceil.$$

Whenever the algorithm asks "is $u$ adjacent to $v$?" the daemon answers `yes` if $u$ and $v$ belong to the same subset, i.e.

- `yes` if $u, v \in X$ or $u, v \in Y$.
- `no` if $u \in X, v \in Y$ or $u \in Y, v \in X$.

In other words, the daemon answers as if $G$ consists of the disjoint union of two complete graphs on $X$ and $Y$ respectively.
(A graph is called **complete** if each pair of distinct vertices are joined by exactly one edge.)

Now suppose the algorithm halts and returns an answer (connected or disconnected) after asking fewer than \( M = \lceil \sqrt{\frac{1}{2}} \rceil \) questions. There must then exist a pair \( x \in X \) and \( y \in Y \) about which the algorithm has not inquired.

If the algorithm says \( G \) is connected, the 

**Diagram**

[Diagram not shown, but implied as a pair of complete graphs not connected to each other]

This graph is disconnected and is clearly consistent with the diagnostic sequence of answers.
On the other hand, if the algorithm says G is disconnected, the daemon can claim that G actually consists of two complete graphs on X and Y, with a single edge \( e = xy \) added.

\[
\begin{array}{c}
\text{complete} \quad \text{complete} \\
\circ x \quad \circ \circ y \\
X \quad Y
\end{array}
\]

This graph is connected and is consistent with all the daemon's answers, since the edge \( e = xy \) was not probed by the algorithm.

In either case, the daemon can claim the algorithm is wrong. Thus any algorithm which does not ask at least \( \lceil \sqrt{n/2} \rceil^2 \geq \sqrt{2n^2} \) questions (in worst case) cannot be correct.

Thus DFS cannot be improved upon, except perhaps for improvements in hidden constants.
Note: A complete graph on \( n \) vertices has \( \binom{n}{2} = \frac{n(n-1)}{2} \) edges, since each edge corresponds to a unique 2-element subset of \( V(G) \).

**Exercise:**

Give a more sophisticated adversary argument showing that any "adjacency based" algorithm to determine connectivity must ask at least \( \binom{n}{2} \) questions (in worst case.) **Note:** \( \binom{n}{2} = \left\lfloor \frac{n^2}{2} \right\rfloor \leq \frac{n^2}{2} \leq \frac{n^2}{2} \).

In other words, a correct algorithm must inquire about each of the \( \binom{n}{2} \) potential edges.

**Exercise**

Use an adversary argument to show that \( \binom{n}{2} \) "adjacency" questions are necessary (in worst case) to determine if a graph \( G \) is acyclic.

**Exercise**

Give an adversary argument showing that a comparison sort must do at least \( \lceil n \ln n \rceil \) comparisons in worst case on input of size \( n \).
(Note: This is essentially no different from the adversary lower bound for 20 questions since sorting n elements is really a search of n! permutations. The adversary must answer in a way which keeps the pool of candidate permutations as large as possible.)

**Problem**

Let \( b = b_1, b_2, b_3, b_4, b_5 \) be a bit string of length 5. Determine whether or not \( b \) contains the substring \( 111 \) (i.e., 3 consecutive 1's).

Consider algorithms whose only allowed operation is to peek at a bit.

Obviously 5 peeks are sufficient. A decision tree argument provides the (useless) fact that at least 1 peak is necessary.

**Exercise:**

a) Use an adversary argument to show that 4 peeks are necessary in general.

b) Design an algorithm which solves the problem in only 4 peeks.