CSE 102
Analysis of Algorithms
Lower Bounds and Computational Complexity

Consider some problem *P*, in all its instances. Let *n* denote the size of an instance of *P*. Our goals are twofold.

1. Determine an algorithm that solves *P*, and find an asymptotic upper bound $O\big(f(n)\big)$ on its runtime, for some function $f(n)$. We aim to reduce $f(n)$ by discovering better and better algorithms for *P*.

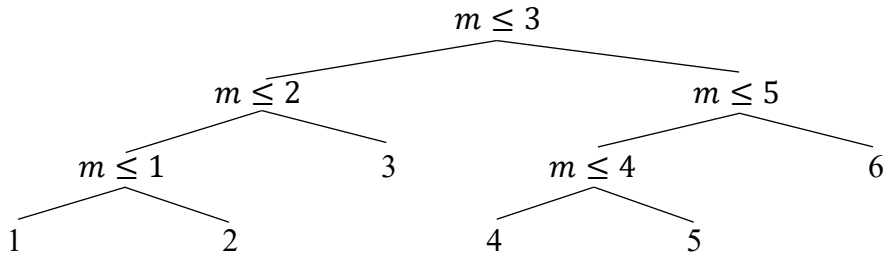
2. Prove that any algorithm solving *P* runs in time $\Omega\big(g(n)\big)$ for some function $g(n)$. We aim to increase $g(n)$ as far as possible by discovering finer and finer proofs.

We are happy when $f(n) = \Theta\big(g(n)\big)$, for then we know we have the best possible algorithm for *P*, apart from improvements in constants hidden in the asymptotic notation. (1) is called *Algorithmics* by some authors, which means the systematic design and analysis of algorithms for solving specific problems. (2) is the theory of *Computational Complexity*, and it considers all possible algorithms for a given problem *P*. The function $g(n)$ is called a *lower bound* on the complexity of *P*.

Decision Tree Lower Bounds

Example Let *m* be an element of the set $\{1, 2, 3, 4, 5, 6\}$ and consider the following problem. Determine *m* by asking a sequence of yes/no questions.

Since this is in essence a searching problem, a moment's thought suggests some variation on Binary Search should be used. We depict an algorithmic solution to this problem by displaying a *decision tree*.

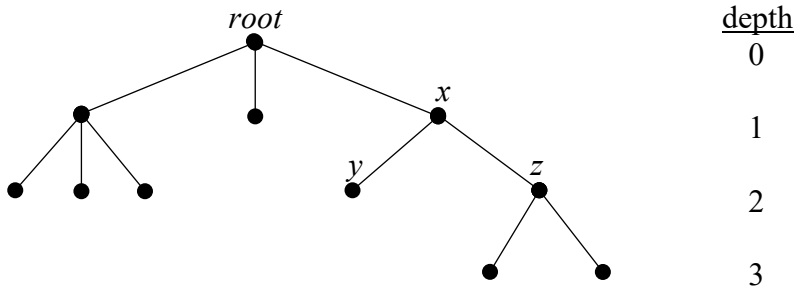


In this tree, each branch to the left indicates the answer yes, and a branch to the right represents the answer no. Each leaf corresponds to a verdict in the set $\{1, 2, 3, 4, 5, 6\}$. From this tree, one can easily see that the worst case number of questions is 3. Decision tree representations are particularly good for finding average case run times. Since there are 4 verdicts at depth 3, and 2 verdicts at depth 2, the average number of questions is $\frac{4\cdot 3+2\cdot 2}{6} = \frac{8}{3} = 2.66$. Apparently the correct answer can be obtained by asking no more than 3 questions. It seems obvious that in general, 2 questions will not suffice, and indeed this is not hard to prove. (There are only 4 combinations of yes/no answers that can be given to a sequence of 2 questions: (yes, yes), (yes, no), (no, yes) and (no, no), not enough to distinguish all 6 possible verdicts.)

A *decision tree argument* is a general technique that gives a lower bound on the complexity of a problem *P*, by reasoning about the possible decision tree representations of an algorithm that solves *P*. The

preceding example suggests it is worthwhile to first collect a few combinatorial facts regarding rooted trees. The reader who is unfamiliar with graph theory should first review the handout on that topic.

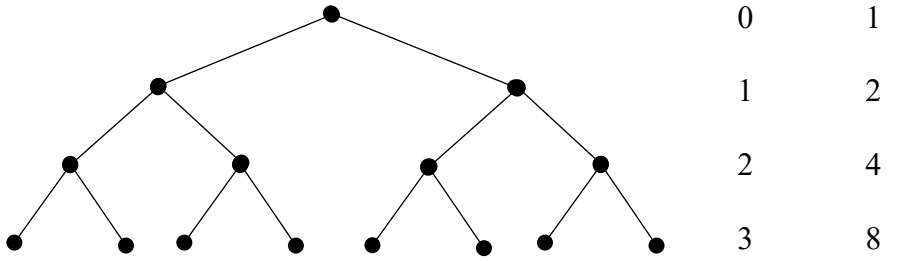
Definitions A *rooted tree* is a tree in which one vertex (*node* in this context) has been distinguished as the *root*. The *depth* of a node is its distance from the root. The *parent* of a node *y* is the (unique) node *x* which is adjacent to *y* and has depth 1 less than that of *x*. A *child* of *x* is any node having *x* as parent.



Note the root, having depth 0, is the only node with no parent. If a node (such as *y* in the above figure) has no children, it is called a *leaf*. A non-leaf is called an *internal node*. The *height* of a rooted tree is its maximum node depth, i.e. the length of longest path from root to a descendant leaf. The *height* of a node is the height of the subtree rooted at that node. Thus in the above figure, the height of the tree is 3, while the height of node *x* is 2. The height of a leaf is necessarily 0.

A *k-ary tree* is a rooted tree in which each node has at most *k* children. The term *binary tree* usually refers to something more specific than a 2-ary tree. In a binary tree, each child is identified as either *left child* or *right child* of its respective parent.

A *complete binary tree* (CBT) is a binary tree in which all leaves are at the same depth, and every internal node has exactly two children. The following CBT has height $h = 3$.



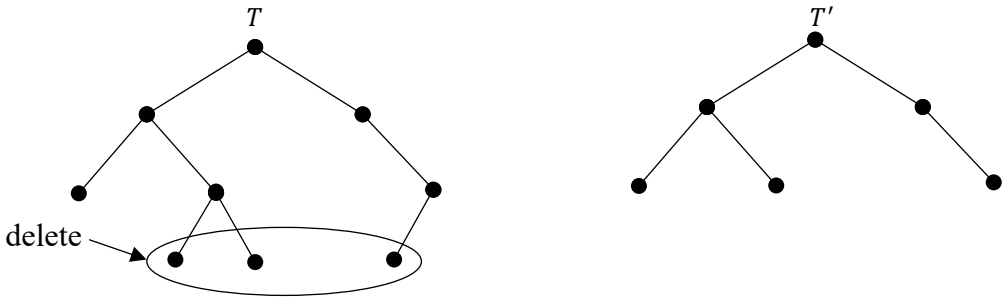
Observe that in a CBT, the number of nodes at depth *d* is 2^d. Therefore the number of leaves is 2^h, where h is the height of the tree. We see that the height of a CBT with *n* leaves is $h = \lg(n)$. Complete binary trees are very rare within the set of all binary trees however. They have the minimum possible height amongst all binary trees with the same number of leaves.

Theorem Let *T* be a binary tree with *n* leaves and height *h*. Then necessarily $h \geq \lceil \lg n \rceil$.

Proof:
Let $L(T)$ and $H(T)$ denote the number of leaves, and the height (respectively) of a binary tree *T*. We proceed by induction on $h = H(T)$.

I. If $h = 0$, then T contains just one node (the root), which is also a leaf. Thus $n = L(T) = 1$, and the inequality $h \geq \lceil \lg n \rceil$ reduces to $0 \geq 0$. The base case is therefore satisfied.

II. Let $h > 0$, and assume for any a binary tree T' with $H(T') = h - 1$, that $H(T') \geq \lceil \lg(L(T')) \rceil$. We must show that $H(T) \geq \lceil \lg(L(T)) \rceil$, i.e. $h \geq \lceil \lg(n) \rceil$. Let T' be the binary tree obtained by deleting from *T*, all leaves at depth *h* (along with all incident edges.)



Observe then that $H(T') = h - 1$, and so by the induction hypothesis $H(T') \geq \lceil \lg(L(T')) \rceil$. Since each node in *T* has at most 2 children, we also have $L(T) \leq 2L(T')$, and hence $L(T') \geq L(T)/2$. Putting these inequalities together, we get

$$\begin{aligned} h - 1 &= H(T') \\ &\geq \lceil \lg(L(T')) \rceil && \text{(by the induction hypothesis)} \\ &\geq \lceil \lg(L(T)/2) \rceil \\ &= \lceil \lg(L(T)) - 1 \rceil \\ &\geq \lceil \lg(n) \rceil - 1 && \text{(if } u \in \mathbb{R} \text{ and } a \in \mathbb{Z}\text{, then } \lceil u \pm a \rceil = \lceil u \rceil \pm a\text{)} \end{aligned}$$

and therefore $h \geq \lceil \lg(n) \rceil$, as required. ■

Exercise Let *T* be a *k*-ary tree with *n* leaves and height *h*. Prove that $h \geq \lceil \log_k n \rceil$.

To use *k*-ary trees to find lower bounds, we reason as follows. Given a problem *P*, consider all algorithms that solve *P* by performing a sequence of basic operations, each of which results in one of (at most) *k* possible outcomes. We call these operations *k-ary probes*, or just *probes* if *k* is understood. Let *n* denote the size of an instance of *P*, and let $f(n)$ be the number of possible algorithm outputs (*verdicts*) for such an instance. Any algorithm of this kind can be represented as a *k*-ary decision tree. Each internal node represents a probe of the input data, and each of its (at most *k*) children represents one of the *k* outcomes to that probe. Each leaf represents an output of the algorithm, and so every downward path from the root to a descendant leaf represents a particular sequence of probes leading to a particular verdict. The height of the tree is therefore the maximum number of probes needed to reach a verdict, i.e. the worst case runtime. Note that there may be more than one path to the same verdict, so in general there may be more leaves than verdicts. For the algorithm to work however, there cannot be more verdicts than leaves. Therefore, if *T* is a decision tree for such an algorithm, then $f(n) \leq L(T)$. By the preceding exercise, the height of *T* satisfies $h \geq \lceil \log_k L(T) \rceil \geq \lceil \log_k f(n) \rceil$. We have proved the following.

Theorem
Suppose a problem *P* has $f(n)$ possible outputs on input of size *n*. Then no algorithm for *P* that uses only *k*-ary probes of the input data can perform fewer than $\lceil \log_k f(n) \rceil$ such probes on input of size *n*, in worst case. Thus $\lceil \log_k f(n) \rceil$ is a lower bound for the (worst case) runtime of such an algorithm.

It is important to remember that this theorem does not assert the existence of an algorithm that solves *P* in only $\lceil \log_k f(n) \rceil$ steps. Instead, it asserts the *non-existence* of any algorithm that does fewer. Also note that the theorem is restricted to the class of algorithms doing only *k*-ary probes. Changing k changes the lower bound. Since all log functions are asymptotically equivalent, the theorem implies a single asymptotic lower bound $\Omega(\log f(n))$ for any *k*.

Example To generalize our earlier guessing game example, let $S = \{1, 2, 3, \dots, n\}$ and consider the problem of finding a target $m \in S$ by asking a sequence of questions, each with at most *k* answers. We have $f(n) = n$ possible verdicts, and therefore any valid algorithm for this problems asks at least $\lceil \log_k n \rceil$ questions in worst case. If $n = 6$ and $k = 2$, then $\lceil \lg 6 \rceil = 3$, so 2 questions will not suffice. If $n = 1{,}000{,}000 = 10^6$ and $k = 2$, then $\lceil \lg 10^6 \rceil = 20$, which explains where the game "twenty questions" gets its name. If $n = 10^6$ and $k = 3$ (so perhaps we ask to which of 3 subsets *m* belongs), then $\lceil \log_3 10^6 \rceil = 13$. Again this does not show that 13 questions will suffice, only that 12 will not.

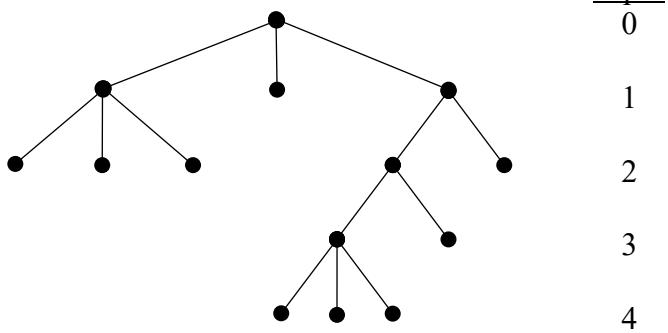
Theorem
Any comparison based sorting algorithm must do, in worst case, at least $\lceil \lg(n!) \rceil$ comparisons on input arrays of length *n*. This gives us an asymptotic lower bound of $\Omega(n \log n)$ for comparison sorts.

Proof:
A verdict for a given input array $A[1 \cdots n]$ is a re-arrangement of the array, of which there are $n!$, and therefore $f(n) = n!$. Each comparison $A[i] \leq A[j]$ (or $A[i] < A[j]$) has one of 2 possible outcomes, true or false. Therefore the worst case number of comparisons is at least $\lceil \lg(n!) \rceil$. By Stirling's formula $\lceil \lg(n!) \rceil = \Omega(\log(n!)) = \Omega(n \log n)$. ■

Decision trees can also be used to find the average case complexity of a problem.

Definition
The *average height* of a *k*-ary tree is its average leaf depth. (Note: we make the default assumption that each leaf is equally weighted in this average.) Thus if *T* has *n* leaves at depths $d_1, d_2, \dots, d_n$, then the average height of *T* is $a = (\sum_{i=1}^{n} d_i)/n$.

Example $k = 3,\ n = 9$



$$a = \frac{1+2+2+2+2+3+4+4+4}{9} = \frac{24}{9} = \frac{8}{3} = 2.66\ldots$$

Just as the height of a decision tree gives the worst case number of probes performed by any algorithm it represents, the average height gives the average number of probes, assuming that each execution path to a verdict is equally likely to be performed.

Theorem
The average height a of a k-ary tree having n leaves satisfies $a \geq \log_k n$.

See p.416 of Brassard & Bratley for a proof of this theorem in the case $k = 2$. Applying this to decision trees that represent comparison sorts, we obtain the following.

Theorem
Any comparison based sorting algorithm must do, in average case, at least $\lg(n!)$ comparisons on input arrays of length n. Again Stirling's formula yields the asymptotic lower bound $\Omega(n \log n)$ for the average case runtime of a comparison sort.

To summarize, the decision tree method for finding lower bounds on the runtime of a problem P is a very simple and straightforward process.

(1) Determine k, the maximum number of outcomes to each probe of the input data.
(2) Determine $f(n)$, the number of possible verdicts (algorithm outputs) as a function of the input size n.
(3) Conclude that any algorithm for P that uses only probes of the kind analyzed in (1) must do at least $\lceil \log_k f(n) \rceil$ probes (worst case), or $\log_k f(n)$ probes (average case) on input of size n.

As usual, these arguments do not assert the existence of algorithms that perform the number of probes stated above, only the non-existence of algorithms doing fewer.

Adversary Arguments

Example Consider the guessing game discussed earlier, except now one of the two players cheats. Call the players A and B, respectively. Player A pretends to choose a number $x \in \{1, 2, 3, \ldots, n\}$. Player B, who is honest, asks a sequence $Q_1, Q_2, Q_3, \ldots$ of yes/no questions. Player A always gives an answer that is consistent with all previous answers, but which is designed to prolong the game as far as possible. For A's answers to be consistent, there must always exist an $x \in \{1, 2, 3, \ldots, n\}$ such that if x had been picked (by an honest player), the true answers would have been those given by A. At some point, the set of remaining candidates for x reduces to just one number, and player A commits to that number, ending the game. Player B just appears to be unlucky, but there is no proof that A cheated. After all, the number supposedly "picked" by A could have been that final number.

How long can A keep this up? The answer to this question provides a lower bound on the worst case number of probes that must be performed by any algorithm playing the role of B. In order that this question be meaningful, we must specify exactly how A calculates his sequence of answers.

Continuing with our example, let S_i denote the set of remaining candidates for x after the i^{th} question has been answered. Initially $S_0 = \{1, 2, 3, \ldots, n\}$. Let Q_i denote the i^{th} question, and let $A_i(x)$ denote the (true) answer to Q_i, if the mystery number is x. So for instance, if $n = 100$ and $Q_1 =$ "is $x \leq 50$?", then $A_1(40) =$ "yes" and $A_1(60) =$ "no". We define for $i \geq 1$ the following sets.

$$Y_i = \{x \in S_{i-1} \mid A_i(x) = \text{yes}\},$$

and

$$N_i = \{x \in S_{i-1} \mid A_i(x) = \text{no}\}.$$

Again with $n = 100$ and $Q_1 =$ "is $x \leq 50$?", we have $Y_1 = \{1, 2, \ldots, 50\}$ and $N_1 = \{51, 52, \ldots, 100\}$. Observe that $Y_i \cap N_i = \emptyset$ and $Y_i \cup N_i = S_{i-1}$. Thus $|Y_i| + |N_i| = |S_{i-1}|$, and by the pigeonhole principle, at least one of Y_i or N_i must contain at least $\left\lceil \frac{|S_{i-1}|}{2} \right\rceil$ numbers.

We can now specify A's strategy precisely: always answer Q_i in a way that implies x is in the larger of the two sets Y_i or N_i. In other words, answer "yes" if $|Y_i| \geq |N_i|$, and "no" otherwise. Thus

$$S_i = \begin{cases} Y_i & \text{if } |Y_i| \geq |N_i| \\ N_i & \text{if } |Y_i| < |N_i| \end{cases},$$

and therefore $|S_i| \geq \left\lceil \frac{|S_{i-1}|}{2} \right\rceil$ for $i \geq 1$. To simplify notation let $b_i = |S_i|$, so $b_i \geq \lceil b_{i-1}/2 \rceil$ for $i \geq 1$. Then

$$\begin{aligned} b_0 &= n \\ b_1 &\geq \lceil b_0/2 \rceil = \lceil n/2 \rceil \\ b_2 &\geq \lceil b_1/2 \rceil \geq \left\lceil \frac{\lceil n/2 \rceil}{2} \right\rceil = \left\lceil \frac{n}{2^2} \right\rceil \\ b_3 &\geq \left\lceil \frac{n}{2^3} \right\rceil \\ &\vdots \\ b_i &\geq \left\lceil \frac{n}{2^i} \right\rceil \\ &\vdots \end{aligned}$$

The process must end when $b_i = 1$, which implies $\lceil n/2^i \rceil = 1$, and therefore $i = \lceil \lg n \rceil$. (Exercise: check directly that $\left\lceil \frac{n}{2^{\lceil \lg n \rceil}} \right\rceil = 1$ but $\left\lceil \frac{n}{2^{\lceil \lg n \rceil - 1}} \right\rceil = 2$.) Thus if player B claims to know the mystery number x after asking no more than $\lceil \lg n \rceil - 1$ questions, then player A can claim at least one other number as his true pick, without contradicting any of his prior answers. It follows that any algorithm playing the part of B must ask at least $\lceil \lg n \rceil$ questions, in worst case.

Exercise Modify this argument to prove that if B is restricted to asking k-ary questions, then at least $\lceil \log_k n \rceil$ such questions are necessary, in worst case.

Notice that we've reached the very same lower bound(s) for the guessing game problem as we did with the decision tree method. Most readers would probably agree that, at least in this example, the adversary argument is more complicated than the decision tree approach. However, we shall see in subsequent examples that adversary arguments often give much better (i.e. higher) lower bounds.

We now summarize the adversary method for finding a lower bound on the runtime of any algorithm for a given problem P.

(1) Suppose that such an algorithm is run against a malevolent *adversary* (also called a *daemon*) that simulates an instance of P of size n.
(2) Whenever the algorithm probes the input data, the daemon answers its probes in a way that is designed to make the algorithm work hard. The daemon is consistent, in that there must always exist at least one instance of P (of size n) that would elicit the daemon's sequence of answers. The daemon's strategy for prolonging the runtime must be specified exactly, as an algorithm. Note that we are not concerned with the runtime of the daemon's algorithm.

(3) Prove that there is a number $h(n)$ (a function of the input size n) with the following property: if the algorithm were to halt and produce an output after fewer than $h(n)$ probes, then there exists at least one instance of P (of size n) that is consistent with all the daemon's answers, but whose correct solution is different from the algorithm output.
(4) Conclude that $h(n)$ is a lower bound for the worst case number of probes that must be performed by any algorithm for P.

Just as with the decision tree method, the above procedure does not prove the existence of an algorithm that solves P in at most $h(n)$ probes. Instead it proves the non-existence of any algorithm that can solve P using no more than $h(n) - 1$ probes.

Example Consider the following easy problem. Given an array $A[1 \cdots n]$ of numbers, find its maximum entry, and the index where the maximum is located. The obvious algorithm for this problem does $n - 1$ comparisons.

```
FindMax(A, n)
1. max = A[1]
2. imax = 1
3. for i = 2 to n
4.       if A[i] > max
5.             max = A[i]
6.             imax = i
7. return (max, imax)
```

The decision tree lower bound says at least $\lceil \lg n \rceil$ comparisons are necessary. (Exercise: confirm this by carrying out the decision tree procedure on this problem.) However, no known algorithm is better than the above, and $n - 1$ comparisons is much worse than $\lceil \lg n \rceil$. (Exercise: draw a decision tree for the operation of FindMax() in the case $n = 4$. Observe that its height is 3, not $\lceil \lg 4 \rceil = 2$, since the same verdict appears on multiple leaves of the tree.)

This is exactly the situation in algorithm analysis with which we are unhappy. We must find either a better (faster) algorithm, or a better (higher) lower bound. An adversary argument provides an improved lower bound.

Consider any algorithm for this problem, and suppose it is run against an adversary simulating an array A of length n. The daemon's strategy is to answer each question concerning a comparison as if $A[i] = i$ for $1 \le i \le n$, i.e. as if $A = (1, 2, 3, \ldots, n)$. 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 halts after doing fewer than $h(n) = n - 1$ comparisons, giving the output $(A[k], k)$, i.e. the algorithm claims that $A[k]$ is maximum in array A.

Let j be an integer in the range $1 \le j \le n$ such that $j \ne k$ and j has not lost any comparisons. Such an integer must exist since, by assumption, at most $n - 2$ comparisons have been performed, and each comparison creates at most one new loser. Therefore there are at most $n - 2$ losers, and hence at least

two non-losers. At this point, the daemon can prove that the algorithm is incorrect by claiming that array A is given by

$$A = \begin{cases} i & \text{if } i \neq j \\ n+1 & \text{if } i = j \end{cases}$$

Indeed, $A[k] = k$ is *not* maximum in this array, and yet the daemon's answers are all consistent with it, since j never lost a comparison. We conclude that no correct algorithm for this problem can perform fewer than $h(n) = n - 1$ comparisons, so our best known algorithm cannot be improved upon.

Example Let $G = (V, E)$ be a graph on $|V| = n \geq 2$ vertices and consider the following problem: determine whether or not G is connected. We consider algorithms that only ask questions of the form "is vertex u adjacent to vertex v?". We call these *adjacency questions* or sometimes *edge probes*.

The decision tree lower bound is trivial: $k = \#\text{outcomes per question} = 2$ (yes/no), and $f(n) = \#\text{verdicts} = 2$ (connected/disconnected). We conclude that at least $\lceil \lg 2 \rceil = 1$ adjacency question is necessary. This lower bound is far too low to be of any use.

Consider any algorithm based on adjacency questions, and run it against an adversary simulating a graph $G = (V, E)$ with $|V| = n \geq 2$ vertices. The daemon first partitions V into two subsets X and Y of sizes $\lfloor n/2 \rfloor$ and $\lceil n/2 \rceil$ respectively. Thus

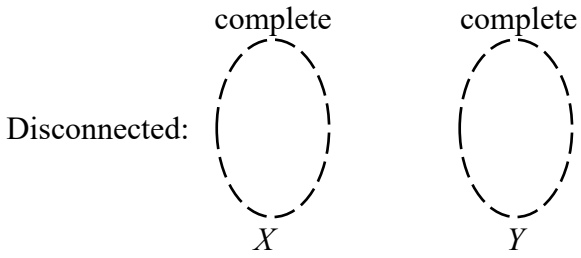
$$X \cup Y = V, \quad X \cap Y = \emptyset, \quad |X| = \lfloor n/2 \rfloor \quad \text{and} \quad |Y| = \lceil n/2 \rceil.$$

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

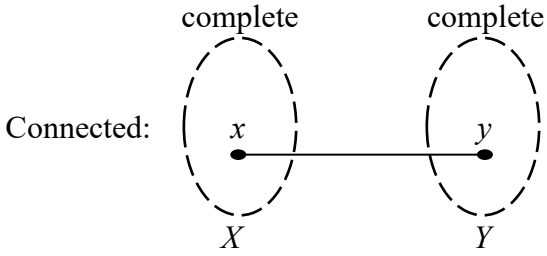
$$\begin{cases} \text{yes} & \text{if } u, v \in X \text{ or } u, v \in Y \\ \text{no} & \text{if } u \in X, v \in Y \text{ or } u \in Y, v \in X \end{cases}$$

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* iff each pair of distinct vertices are joined by exactly one edge.) Suppose the algorithm halts and returns an output (connected/disconnected) after asking fewer than $h(n) = \lfloor n/2 \rfloor \cdot \lceil n/2 \rceil$ adjacency questions. Then there must exist a pair of vertices $x \in X$ and $y \in Y$ about which the algorithm has not asked.

If the algorithm says G is connected, then the daemon can claim, to the contrary, that G consists of two disjoint complete graphs on X and Y.



On the other hand, if the algorithm says that G is disconnected, then the daemon can claim that G consists of the above graph, with a single edge $e=xy$ added.



Since $\{x, y\}$ was not probed, both of the above graphs are consistent with all answers given by the daemon, and yet one is connected while the other is not. No matter what is the algorithm's verdict, the daemon can show it to be wrong. Thus any algorithm that does not ask at least $\lfloor n/2 \rfloor \cdot \lceil n/2 \rceil$ adjacency questions cannot be correct. Therefore $h(n) = \lfloor n/2 \rfloor \cdot \lceil n/2 \rceil$ is a lower bound on the run time of such an algorithm.

Note that Depth-First Search (DFS) can solve this problem in time $\Omega(n^2)$, which matches the above result asymptotically. (This assumes a graph is represented by its *adjacency matrix*. The run time analysis of DFS is slightly different if the *adjacency list* representation is used. See section 22.3 in CLRS and the handout on graph theory.) Observe that the number of edges in K_n, the complete graph on n vertices, is $\binom{n}{2} = \frac{n(n-1)}{2}$, since this is the number of 2-element subsets of an n element set. This is therefore the maximum possible number of edges in a graph with n vertices. It seems reasonable to expect that one must probe every potential edge in a graph to determine connectivity. Both $\binom{n}{2}$ and $\lfloor n/2 \rfloor \cdot \lceil n/2 \rceil$ are in the class $\Omega(n^2)$. As one can see however, $\lfloor n/2 \rfloor \cdot \lceil n/2 \rceil \sim \frac{1}{4}n^2$ while $\binom{n}{2} \sim \frac{1}{2}n^2$, so $\binom{n}{2}$ has the better (higher) lead coefficient. The following theorem proves this better lower bound, using a more delicate adversary argument.

Theorem At least $\binom{n}{2}$ adjacency questions are necessary (in worst case) to determine whether a graph G on n vertices is connected.

Proof: Consider any algorithm for this problem, and run it against the following adversary simulating an input graph G with n vertices. The daemon's strategy is to answer *no* to all edge probes, unless that answer would prove G is disconnected. More precisely, the daemon maintains two edge sets A and B, where initially B is empty and A contains all $\binom{n}{2}$ edges in K_n, the complete graph on n vertices. The Daemon then performs the following algorithm when an edge e is probed:

Probe(e)
1. if $A - e$ is connected
2. $\quad A = A - e$
3. $\quad$ answer No
4. else
5. $\quad B = B + e$
6. $\quad$ answer Yes

We abuse notation slightly and identify the edge set A with the subgraph $(\{1, 2, \ldots, n\}, A)$ of K_n consisting of all vertices in K_n together with the edges in A (and likewise for B.) Observe that at all times $B \subseteq A$, and $A - B$ consists of exactly those edges of K_n not yet probed. Furthermore, both A and B are consistent with the daemon's sequence of answers. Indeed, whenever the answer Yes is given for

an edge, that edge is added to B and remains in A. If No is given, the corresponding edge is removed from A and is not added to B. The following invariants are maintained over any sequence of edge probes.

(a) The subgraph A is always connected. This is obvious from the construction.
(b) If A contains a cycle, then none of its edges belong to B. (Proof: deleting an edge from a cycle in A would leave A connected, so that edge could not have been added to B.)
(c) It follows from (b) that B is acyclic.
(d) If $A \neq B$, then B is disconnected. (Proof: assume, to get a contradiction, that B is connected. Then being acyclic, B is a tree. Since $A \neq B$ and $B \subseteq A$, there exists an edge $e \in A - B$. If e were added to B, it would form a cycle with some of the other edges in B. (This is a well-known property of trees: joining vertices by a new edge creates a unique cycle.) Since $B \subseteq A$, that cycle is also contained in A. In other words, A contains a cycle consisting of e together with some edges in B, contradicting (b) above. The only way to avoid the contradiction is to conclude that B is disconnected.

Suppose the algorithm halts and returns a verdict (connected/disconnected) after probing fewer than $\binom{n}{2}$ edges. Then at least one edge of K_n was not probed, hence $A - B \neq \emptyset$ and $A \neq B$. Remark (d) now says B is disconnected, while (a) assures us that A is connected. But both graphs are consistent with the daemon's sequence of answers, so if the algorithm says G is connected, then the daemon can claim $G = B$, while if the algorithm says G is disconnected, the daemon may insist $G = A$. In either case the algorithm is shown to be incorrect. We conclude that any adjacency based algorithm that correctly establishes connectivity, must probe all $\binom{n}{2}$ potential edges. ■

Exercise Show that at least $\binom{n}{2}$ "adjacency" questions are necessary to determine whether a graph G on n vertices is acyclic. (Hint: use the following adversary strategy. Answer yes to any edge probe, unless that answer would prove the existence of a cycle.)

Exercise Let $b = x_1x_2x_3x_4x_5$ be a bit string of length 5, i.e. $x_i \in \{0, 1\}$ for $1 \leq i \leq 5$. Consider the following problem. Determine whether or not b contains the substring 111. We restrict attention to those algorithms whose only operation is to peek at a bit. Obviously 5 peeks are sufficient. A decision tree argument provides the (useless) fact that at least one peek is necessary.

a. Design an algorithm for this problem that uses only 4 peeks in worst case. Express your algorithm as a decision tree.

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

Example
As we saw in an earlier homework problem, it is possible to determine (min, max) of an array of length n by doing $\lceil 3n/2 \rceil - 2$ comparisons. The question naturally arises as to whether it is possible to do any better. We now show by an adversary argument that the answer is no. We leave as an exercise to prove the (inferior) decision tree lower bound of $\lceil \lg(n^2 - n + 1) \rceil = \Omega(\log(n))$.

Theorem
Any algorithm that performs only array comparisons must do at least $\lceil 3n/2 \rceil - 2$ such comparisons in order to determine both the minimum and maximum of an array of length n.

Proof:
Let any algorithm run against the following adversary, which simulates an input array of length n. The daemon first marks each array element with two symbols: $+$ and $-$. A $+$ signifies that the element is a candidate for the maximum, and a $-$ sign denotes a candidate for minimum. Thus initially, the array is adorned with $2n$ marks.

Each comparison $A[i] < A[j]$ requires that the daemon commit one element as smaller (the loser) and one as larger (the winner). A loser can no longer be considered a candidate for maximum, and a winner is eliminated from consideration as the minimum. Therefore, as the algorithm continues to probe the array, the daemon removes marks as appropriate to its answers. At all times then, each array element will therefore be in one of the following 4 states.

$\pm$	candidate for both max and min
$+$	candidate for max only
$-$	candidate for min only
N (no mark)	candidate for neither max nor min

The daemon's answer will depend on the current markings of the elements being compared The marks are then changed to be consistent with the answer given. For instance, when two elements marked $\pm$ are compared, one element is randomly selected as winner and the other as loser. The daemon erases the $-$ sign from the winner, and erases the $+$ sign from the loser. When two elements each marked $+$ are compared, a winner and loser are chosen (randomly), and the loser has its $+$ removed. When a $+$ is compared to an N (no mark), the $+$ is reported as winner, the N as loser, and no change is made to the marking. Note that in this process, $+$ and $-$ signs are always retained or removed, but never added. The following table gives all possible types of comparisons, the corresponding actions by the daemon, and the number of marks removed in each case. The answer given by the daemon is understood to be consistent with the new (or unchanged) marking.

Comparison	Action	# marks removed
$\pm$ to $\pm$	randomly replace one $\pm$ by $+$, the other by $-$	2
$\pm$ to $+$	retain $+$, change $\pm$ to $-$	1
$\pm$ to $-$	retain $-$, change $\pm$ to $+$	1
$\pm$ to N	randomly replace $\pm$ by either $+$ or $-$	1
$+$ to $+$	randomly chose one $+$ to become N	1
$-$ to $-$	randomly chose one $-$ to become N	1
$+$ to $-$	make no changes	0
$+$ to N	make no changes	0
$-$ to N	make no changes	0
N to N	make no changes	0

Let

$c_0 =$ the # of comparisons that remove no marks
$c_1 =$ the # of comparisons that remove 1 mark
$c_2 =$ the # of comparisons that remove 2 marks

Assume that the algorithm halts and returns a verdict after performing fewer than $\lceil 3n/2 \rceil - 2$ comparisons. We leave it as an exercise to show that $\lceil 3n/2 \rceil - 2 = 2n - 2 - \lfloor n/2 \rfloor$, and therefore

$$\begin{aligned} & c_0 + c_1 + c_2 < 2n - 2 - \lfloor n/2 \rfloor \\ \Rightarrow\ & c_1 + 2c_2 < (2n - 2) + (c_2 - \lfloor n/2 \rfloor) - c_0 \\ \Rightarrow\ & c_1 + 2c_2 < 2n - 2 \end{aligned}$$

The last inequality follows since $c_0 \geq 0$, and no more than $\lfloor n/2 \rfloor$ comparisons can remove 2 marks, i.e. $c_2 \leq \lfloor n/2 \rfloor$. (There are at most $\lfloor n/2 \rfloor$ pairs of elements from which to remove 2 marks, using the first type of comparison listed in the table above.) The total number of marks removed is therefore $0 \cdot c_0 + 1 \cdot c_1 + 2 \cdot c_2 = c_1 + 2c_2$, and therefore

$$\begin{aligned} & \#\text{ marks removed} < 2n - 2 \\ \Rightarrow\ & \#\text{ marks remaining} > 2n - (2n - 2) = 2 \\ \Rightarrow\ & \#\text{ marks remaining} \geq 3 \end{aligned}$$

With at least 3 marks remaining, there must exist either two $+$ signs or two $-$ signs. Say for instance that two $+$ signs remain. Then whatever element the algorithm says is the maximum, the daemon can claim another viable candidate as the actual maximum. A similar situation holds if there are two $-$ signs remaining, for then the daemon can negate any algorithm output regarding the location of the minimum. Therefore, no correct algorithm can perform fewer than $\lceil 3n/2 \rceil - 2$ array comparisons. ■