An adversary means opposition and competition,
but not having an adversary means grief and loneliness.

— Zhuangzi (Chuang-tsu) c. 300 BC

It is possible that the operator could be hit by an asteroid and your $20 could fall off his cardboard box and land on the ground, and while you were picking it up, $5 could blow into your hand. You therefore could win $5 by a simple twist of fate.

— Penn Jillette, explaining how to win at Three-Card Monte (1999)

29 Adversary Arguments

29.1 Three-Card Monte

Until Times Square was turned into a glitzy sanitized tourist trap, you could often find dealers stealing tourists' money using a game called "Three Card Monte" or "Spot the Lady". The dealer show the tourist three cards, say the Queen of Hearts, the two of spades, and three of clubs. The dealer shuffles the cards face down on a table (usually slowly enough that the tourist can follow the Queen), and then asks the tourist to bet on which card is the Queen. In principle, the tourist's odds of winning are at least one in three, more if the tourist was carefully watching the movement of the cards.

In practice, however, the tourist *never* wins, because the dealer cheats. The dealer actually holds at least *four* cards; before he even starts shuffling the cards, the dealer palms the queen or sticks it up his sleeve. No matter what card the tourist bets on, the dealer turns over a black card (which might be the two of clubs, but most tourists won't notice that wasn't one o the original cards). If the tourist gives up, the dealer slides the queen under one of the cards and turns it over, showing the tourist 'where the queen was all along'. If the dealer is really good, the tourist won't see the dealer changing the cards and will think maybe the queen *was* there all along and he just wasn't smart enough to figure that out. As long as the dealer doesn't reveal all the black cards at once, the tourist has no way to prove that the dealer cheated![1]

29.2 n-Card Monte

Now let's consider a similar game, but with an algorithm acting as the tourist and with bits instead of cards. Suppose we have an array of n bits and we want to determine if any of them is a 1. Obviously we can figure this out by just looking at every bit, but can we do better? Is there maybe some complicated tricky algorithm to answer the question "Any ones?" without looking at every bit? Well, of course not, but how do we prove it?

The simplest proof technique is called an *adversary* argument. The idea is that an all-powerful malicious adversary (the dealer) *pretends* to choose an input for the algorithm (the tourist). When the algorithm wants looks at a bit (a card), the adversary sets that bit to whatever value will make the algorithm do the most work. If the algorithm does not look at enough bits before terminating, then there will be several different inputs, each consistent with the bits already seen,

[1] Even if the dealer is a sloppy magician, he'll cheat anyway. The dealer is almost always surrounded by shills; these are the "tourists" who look like they're actually winning, who turn over cards when the dealer "isn't looking", who casually mention how easy the game is to win, and so on. The shills physically protect the dealer from any angry tourists who notice the dealer cheating, and shake down any tourists who refuse to pay after making a bet. Really, you ***cannot*** win this game, ***ever***.

the should result in different outputs. Whatever the algorithm outputs, the adversary can 'reveal' an input that is has all the examined bits but contradicts the algorithm's output, and then claim that that was the input that he was using all along. Since the only information the algorithm has is the set of bits it examined, the algorithm cannot distinguish between a malicious adversary and an honest user who actually chooses an input in advance and answers all queries truthfully.

For the n-card monte problem, the adversary originally pretends that the input array is all zeros—whenever the algorithm looks at a bit, it sees a 0. Now suppose the algorithms stops before looking at all three bits. If the algorithm says 'No, there's no 1,' the adversary changes one of the unexamined bits to a 1 and shows the algorithm that it's wrong. If the algorithm says 'Yes, there's a 1,' the adversary reveals the array of zeros and again proves the algorithm wrong. Either way, the algorithm cannot tell that the adversary has cheated.

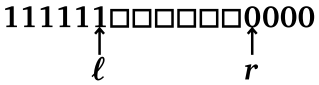
One absolutely crucial feature of this argument is that *the adversary makes absolutely **no** assumptions about the algorithm*. The adversary strategy can't depend on some predetermined order of examining bits, and it doesn't care about anything the algorithm might or might not do when it's not looking at bits. *Any* algorithm that doesn't examine every bit falls victim to the adversary.

29.3 Finding Patterns in Bit Strings

Let's make the problem a little more complicated. Suppose we're given an array of n bits and we want to know if it contains the substring 01, a zero followed immediately by a one. Can we answer this question without looking at every bit?

It turns out that if n is odd, we *don't* have to look at all the bits. First we look the bits in every even position: $B[2], B[4], \ldots, B[n-1]$. If we see $B[i] = 0$ and $B[j] = 1$ for any $i < j$, then we know the pattern 01 is in there somewhere—starting at the last 0 before $B[j]$—so we can stop without looking at any more bits. If we see only 1s followed by 0s, we don't have to look at the bit between the last 0 and the first 1. If every even bit is a 0, we don't have to look at $B[1]$, and if every even bit is a 1, we don't have to look at $B[n]$. In the worst case, our algorithm looks at only $n-1$ of the n bits.

But what if n is even? In that case, we can use the following adversary strategy to show that any algorithm *does* have to look at every bit. The adversary will attempt to produce an 'input' string B *without* the substring 01; all such strings have the form $11\ldots100\ldots0$. The adversary maintains two indices ℓ and r and pretends that the prefix $B[1..\ell]$ contains only 1s and the suffix $B[r..n]$ contains only 0s. Initially $\ell = 0$ and $r = n+1$.



What the adversary is thinking; □ represents an unknown bit.

The adversary maintains the invariant that $r - \ell$, the length of the undecided portion of the 'input' string, is even. When the algorithm looks at a bit between ℓ and r, the adversary chooses whichever value preserves the parity of the intermediate chunk of the array, and then moves either ℓ or r. Specifically, here's what the adversary does when the algorithm examines bit $B[i]$. (Note that I'm specifying the adversary strategy as an algorithm!)

```
HIDE01($i$):
  if $i \le \ell$
      $B[i] \leftarrow 1$
  else if $i \ge r$
      $B[i] \leftarrow 0$
  else if $i - \ell$ is even
      $B[i] \leftarrow 0$
      $r \leftarrow i$
  else
      $B[i] \leftarrow 1$
      $\ell \leftarrow i$
```

It's fairly easy to prove that this strategy forces the algorithm to examine every bit. If the algorithm doesn't look at every bit to the right of r, the adversary could replace some unexamined bit with a 1. Similarly, if the algorithm doesn't look at every bit to the left of ℓ, the adversary could replace some unexamined bit with a zero. Finally, if there are any unexamined bits between ℓ and r, there must be at least two such bits (since $r - \ell$ is always even) and the adversary can put a 01 in the gap.

In general, we say that a bit pattern is *evasive* if we have to look at every bit to decide if a string of n bits contains the pattern. So the pattern 1 is evasive for all n, and the pattern 01 is evasive if and only if n is even. It turns out that the *only* patterns that are evasive for *all* values of n are the one-bit patterns 0 and 1.

29.4 Evasive Graph Properties

Another class of problems for which adversary arguments give good lower bounds is graph problems where the graph is represented by an adjacency matrix, rather than an adjacency list. Recall that the adjacency matrix of an undirected n-vertex graph $G = (V, E)$ is an $n \times n$ matrix A, where $A[i, j] = \big[(i, j) \in E\big]$. We are interested in deciding whether an undirected graph has or does not have a certain *property*. For example, is the input graph connected? Acyclic? Planar? Complete? A tree? We call a graph property *evasive* if we have to look look at all $\binom{n}{2}$ entries in the adjacency matrix to decide whether a graph has that property.

An obvious example of an evasive graph property is *emptiness*: Does the graph have any edges at all? We can show that emptiness is evasive using the following simple adversary strategy. The adversary maintains *two* graphs E and G. E is just the empty graph with n vertices. Initially G is the complete graph on n vertices. Whenever the algorithm asks about an edge, the adversary removes that edge from G (unless it's already gone) and answers 'no'. If the algorithm terminates without examining every edge, then G is not empty. Since both G and E are consistent with all the adversary's answers, the algorithm must give the wrong answer for one of the two graphs.

29.5 Connectedness Is Evasive

Now let me give a more complicated example, *connectedness*. Once again, the adversary maintains two graphs, Y and M ('yes' and 'maybe'). Y contains all the edges that the algorithm knows are definitely in the input graph. M contains all the edges that the algorithm thinks *might* be in the input graph, or in other words, all the edges of Y plus all the unexamined edges. Initially, Y is empty and M is complete.

Here's the strategy that adversary follows when the adversary asks whether the input graph contains the edge e. I'll assume that whenever an algorithm examines an edge, it's in M but not in Y; in other words, algorithms never ask about the same edge more than once.

```
HideConnectedness($e$):
    if $M \setminus \{e\}$ is connected
        remove $(i, j)$ from $M$
        return 0
    else
        add $e$ to $Y$
        return 1
```

Notice that the graphs Y and M are both consistent with the adversary's answers at all times. The adversary strategy maintains a few other simple invariants.

- **Y is a subgraph of M.** This is obvious.
- **M is connected.** This is also obvious.
- **If M has a cycle, none of its edges are in Y.** If M has a cycle, then deleting any edge in that cycle leaves M connected.
- **Y is acyclic.** This follows directly from the previous invariant.
- **If $Y \neq M$, then Y is disconnected.** The only connected acyclic graph is a tree. Suppose Y is a tree and some edge e is in M but not in Y. Then there is a cycle in M that contains e, all of whose other edges are in Y. This violated our third invariant.

We can also think about the adversary strategy in terms of minimum spanning trees. Recall the anti-Kruskal algorithm for computing the *maximum* spanning tree of a graph: Consider the edges one at a time in increasing order of length. If removing an edge would disconnect the graph, declare it part of the spanning tree (by adding it to Y); otherwise, throw it away (by removing it from M). If the algorithm examines all $\binom{n}{2}$ possible edges, then Y and M are both equal to the maximum spanning tree of the complete n-vertex graph, where the weight of an edge is the time when the algorithm asked about it.

Now, if an algorithm terminates before examining all $\binom{n}{2}$ edges, then there is at least one edge in M that is not in Y. Since the algorithm cannot distinguish between M and Y, even though M is connected and Y is not, the algorithm cannot possibly give the correct output for both graphs. Thus, in order to be correct, any algorithm must examine every edge—*Connectedness is evasive!*

29.6 An Evasive Conjecture

A graph property is *nontrivial* is there is at least one graph with the property and at least one graph without the property. (The only trivial properties are 'Yes' and 'No'.) A graph property is *monotone* if it is closed under taking subgraphs — if G has the property, then any subgraph of G has the property. For example, emptiness, planarity, acyclicity, and *non*-connectedness are monotone. The properties of being a tree and of having a vertex of degree 3 are not monotone.

Conjecture 1 (Aanderraa, Karp, and Rosenberg). *Every nontrivial monotone property of n-vertex graphs is evasive.*

The Aanderraa-Karp-Rosenberg conjecture has been proven when $n = p^e$ for some prime p and positive integer exponent e—the proof uses some interesting results from algebraic topology[2]—but it is still open for other values of n.

[2] Let Δ be a contractible simplicial complex whose automorphism group $\mathrm{Aut}(\Delta)$ is vertex-transitive, and let Γ be a vertex-transitive subgroup of $\mathrm{Aut}(\Delta)$. If there are normal subgroups $\Gamma_1 \lhd \Gamma_2 \lhd \Gamma$ such that $|\Gamma_1| = p^\alpha$ for some prime p and integer α, $|\Gamma/\Gamma_2| = q^\beta$ for some prime q and integer β, and Γ_2/Γ_1 is cyclic, then Δ is a simplex.

No, this will not be on the final exam.

There are non-trivial non-evasive graph properties, but all known examples are non-monotone. One such property—'scorpionhood'—is described in an exercise at the end of this lecture note.

29.7 Finding the Minimum and Maximum

Last time, we saw an adversary argument that finding the largest element of an unsorted set of n numbers requires at least $n-1$ comparisons. Let's consider the complexity of finding the largest *and* smallest elements. More formally:

> Given a sequence $X = \langle x_1, x_2, \dots, x_n \rangle$ of n distinct numbers, find indices i and j such that $x_i = \min X$ and $x_j = \max X$.

How many comparisons do we need to solve this problem? An upper bound of $2n-3$ is easy: find the minimum in $n-1$ comparisons, and then find the maximum of everything else in $n-2$ comparisons. Similarly, a lower bound of $n-1$ is easy, since any algorithm that finds the min and the max certainly finds the max.

We can improve both the upper and the lower bound to $\lceil 3n/2 \rceil - 2$. The upper bound is established by the following algorithm. Compare all $\lfloor n/2 \rfloor$ consecutive pairs of elements x_{2i-1} and x_{2i}, and put the smaller element into a set S and the larger element into a set L. if n is odd, put x_n into both L and S. Then find the smallest element of S and the largest element of L. The total number of comparisons is at most

$$\underbrace{\left\lfloor \frac{n}{2} \right\rfloor}_{\text{build } S \text{ and } L} + \underbrace{\left\lceil \frac{n}{2} \right\rceil - 1}_{\text{compute } \min S} + \underbrace{\left\lceil \frac{n}{2} \right\rceil - 1}_{\text{compute } \max L} = \left\lceil \frac{3n}{2} \right\rceil - 2.$$

For the lower bound, we use an adversary argument. The adversary marks each element $+$ if it *might* be the maximum element, and $-$ if it *might* be the minimum element. Initially, the adversary puts both marks $+$ and $-$ on every element. If the algorithm compares two double-marked elements, then the adversary declares one smaller, removes the $+$ mark from the smaller element, and removes the $-$ mark from the larger one. In every other case, the adversary can answer so that at most one mark needs to be removed. For example, if the algorithm compares a double marked element against one labeled $-$, the adversary says the one labeled $-$ is smaller and removes the $-$ mark from the other. If the algorithm compares to $+$'s, the adversary must unmark one of the two.

Initially, there are $2n$ marks. At the end, in order to be correct, exactly one item must be marked $+$ and exactly one other must be marked $-$, since the adversary can make any $+$ the maximum and any $-$ the minimum. Thus, the algorithm must force the adversary to remove $2n-2$ marks. At most $\lfloor n/2 \rfloor$ comparisons remove two marks; every other comparison removes at most one mark. Thus, the adversary strategy forces any algorithm to perform at least $2n-2-\lfloor n/2 \rfloor = \lceil 3n/2 \rceil - 2$ comparisons.

29.8 Finding the Median

Finally, let's consider the *median* problem: Given an unsorted array X of n numbers, find its $n/2$th largest entry. (I'll assume that n is even to eliminate pesky floors and ceilings.) More formally:

> Given a sequence $\langle x_1, x_2, \dots, x_n \rangle$ of n distinct numbers, find the index m such that x_m is the $n/2$th largest element in the sequence.

To prove a lower bound for this problem, we can use a combination of information theory and two adversary arguments. We use one adversary argument to prove the following simple lemma:

Lemma 1. *Any comparison tree that correctly finds the median element also identifies the elements smaller than the median and larger than the median.*

Proof: Suppose we reach a leaf of a decision tree that chooses the median element x_m, and there is still some element x_i that isn't known to be larger or smaller than x_m. In other words, we cannot decide based on the comparisons that we've already performed whether $x_i < x_m$ or $x_i > x_m$. Then in particular no element is known to lie between x_i and x_m. This means that there must be an input that is consistent with the comparisons we've performed, in which x_i and x_m are adjacent in sorted order. But then we can swap x_i and x_m, without changing the result of any comparison, and obtain a different consistent input in which x_i is the median, not x_m. Our decision tree gives the wrong answer for this 'swapped' input. □

This lemma lets us rephrase the median-finding problem yet again.

> Given a sequence $X = \langle x_1, x_2, \dots, x_n\rangle$ of n distinct numbers, find the indices of its $n/2-1$ largest elements L and its $n/2$th largest element x_m.

Now suppose a 'little birdie' tells us the set L of elements larger than the median. This information fixes the outcomes of certain comparisons—any item in L is bigger than any element not in L—so we can 'prune' those comparisons from the comparison tree. The pruned tree finds the largest element of $X \setminus L$ (the median of X), and thus must have depth at least $n/2-1$. In fact, the adversary argument in the last lecture implies that *every* leaf in the pruned tree must have depth at least $n/2-1$, so the pruned tree has at least $2^{n/2-1}$ leaves.

There are $\binom{n}{n/2-1} \approx 2^n/\sqrt{n/2}$ possible choices for the set L. Every leaf in the original comparison tree is also a leaf in exactly one of the $\binom{n}{n/2-1}$ pruned trees, so the original comparison tree must have at least $\binom{n}{n/2-1}2^{n/2-1} \approx 2^{3n/2}/\sqrt{n/2}$ leaves. Thus, any comparison tree that finds the median must have depth at least

$$\left\lceil \frac{n}{2} - 1 + \lg\binom{n}{n/2-1}\right\rceil = \frac{3n}{2} - O(\log n).$$

A more complicated adversary argument (also involving pruning the comparison tree with little birdies) improves this lower bound to $2n - o(n)$.

A similar argument implies that at least $n-k+\left\lceil \lg\binom{n}{k-1}\right\rceil = \Omega((n-k)+k\log(n/k))$ comparisons are required to find the kth largest element in an n-element set. This bound is tight up to constant factors for all $k \le n/2$; there is an algorithm that uses at most $O(n+k\log(n/k))$ comparisons. Moreover, this lower bound is *exactly* tight when $k=1$ or $k=2$. In fact, these are the *only* values of $k \le n/2$ for which the exact complexity of the selection problem is known. Even the case $k=3$ is still open!

Exercises

1. (a) Let X be a set containing an odd number of n-bit strings. Prove that any algorithm that decides whether a given n-bit string is an element of X *must* examine every bit of the input string in the worst case.

   (b) Give a one-line proof that the bit pattern 01 is evasive for all even n.

(c) Prove that the bit pattern 11 is evasive if and only if $n \bmod 3 = 1$.

*(d) Prove that the bit pattern 111 is evasive if and only if $n \bmod 4 = 0$ or 3.

2. Suppose we are given the adjacency matrix of a *directed* graph G with n vertices. Describe an algorithm that determines whether G has a *sink* by probing only $O(n)$ bits in the input matrix. A sink is a vertex that has an incoming edge from every other vertex, but no outgoing edges.

*3. A *scorpion* is an undirected graph with three special vertices: the *sting*, the *tail*, and the *body*. The sting is connected only to the tail; the tail is connected only to the sting and the body; and the body is connected to every vertex except the sting. The rest of the vertices (the head, eyes, legs, antennae, teeth, gills, flippers, wheels, etc.) can be connected arbitrarily. Describe an algorithm that determines whether a given n-vertex graph is a scorpion by probing only $O(n)$ entries in the adjacency matrix.

4. Prove using an adversary argument that acyclicity is an evasive graph property. *[Hint: Kruskal.]*

5. Prove that finding the second largest element in an n-element array requires *exactly* $n - 2 + \lceil \lg n \rceil$ comparisons in the worst case. Prove the upper bound by describing and analyzing an algorithm; prove the lower bound using an adversary argument.

6. Let T be a perfect ternary tree where every leaf has depth ℓ. Suppose each of the 3^ℓ leaves of T is labeled with a bit, either 0 or 1, and each internal node is labeled with a bit that agrees with the *majority* of its children.

   (a) Prove that any deterministic algorithm that determines the label of the root must examine all 3^ℓ leaf bits in the worst case.

   (b) Describe and analyze a *randomized* algorithm that determines the root label, such that the expected number of leaves examined is $o(3^\ell)$. (You may want to review the notes on randomized algorithms.)

*7. UIUC has just finished constructing the new Reingold Building, the tallest dormitory on campus. In order to determine how much insurance to buy, the university administration needs to determine the highest safe floor in the building. A floor is consdered *safe* if ~~a drunk student~~ ***an egg*** can fall from a window on that floor and land without breaking; if the egg breaks, the floor is considered *unsafe*. Any floor that is higher than an unsafe floor is also considered unsafe. The only way to determine whether a floor is safe is to drop an egg from a window on that floor.

   You would like to find the lowest unsafe floor L by performing as few tests as possible; unfortunately, you have only a very limited supply of eggs.

   (a) Prove that if you have only one egg, you can find the lowest unsafe floor with L tests. *[Hint: Yes, this is trivial.]*

(b) Prove that if you have only one egg, you must perform at least L tests in the worst case. In other words, prove that your algorithm from part (a) is optimal. *[Hint: Use an adversary argument.]*

(c) Describe an algorithm to find the lowest unsafe floor using *two* eggs and only $O(\sqrt{L})$ tests. *[Hint: Ideally, each egg should be dropped the same number of times. How many floors can you test with n drops?]*

(d) Prove that if you start with two eggs, you must perform at least $\Omega(\sqrt{L})$ tests in the worst case. In other words, prove that your algorithm from part (c) is optimal.

*(e) Describe an algorithm to find the lowest unsafe floor using k eggs, using as few tests as possible, and prove your algorithm is optimal for all values of k.