CS 161 **Problem Set 1**
Autumn 2017 **Due:** Friday, October 6, 2017 at 3pm on Gradescope.

Style guide and expectations: Please see the "Homework" part of the "Resources" section on the webpage for guidance on what we are looking for in homework solutions. We will grade according to these standards.

Exercises

Exercises should be completed **on your own.**

---

0. **(1 pt.)** Have you thoroughly read the course policies on the webpage?

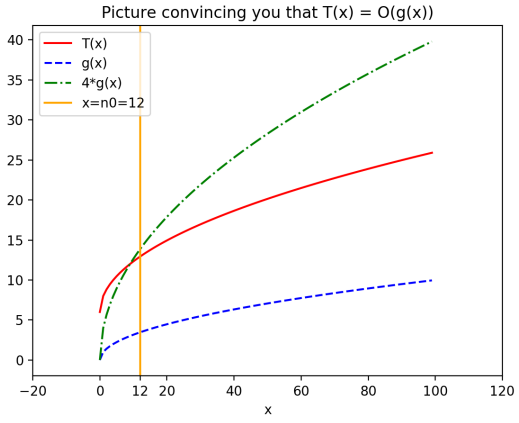
   [We are expecting: The answer "yes."]

   SOLUTION: Yes.

1. **(1 pt.)** See the IPython notebook `HW1.ipynb` for Exercise 1. Modify the code to generate a plot that convinces you that $T(x) = O(g(x))$.[1]

   [We are expecting: Your choice of c, n_0, the plot that you created after modifying the code in Exercise 1, and a short explanation of why this plot should convince a viewer that $T(x) = O(g(x))$.]

   SOLUTION: We choose $c = 4$ and $n_0 = 12$. As we can see in the picture below, for all $n > n_0$ (that is, to the right of the yellow vertical line), we have $cg(n) \geq T(n)$, meaning that the green dashed curve lies above the solid red curve.



2. **(3 pt.)** See the IPython notebook `HW1.ipynb` for Exercise 2, parts A, B and C.

   (A) What is the asymptotic runtime of the function `numOnes( lst )` given in the Python notebook? Give the smallest answer that is correct. (For example, it is true that the runtime is $O(2^n)$, but you can do better).

   [We are expecting: Your answer in the form "The running time of `numOnes( lst )` on a list of size n is $O(\_\_\_)$.", and a few sentences informally justifying why this is the case.]

[1] **Note:** There are instructions for installing Jupyter notebooks in the pre-lecture exercise for Lecture 2.

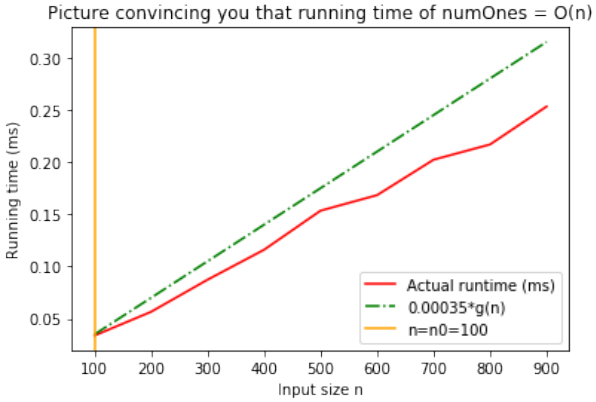
(B) Modify the code in `HW1.ipynb` to generate a picture that backs up your claim from Part (A).
[We are expecting: Your choice of c, n_0, and $g(n)$; the plot that you created after modifying the code in Exercise 2; and a short explanation of why this plot should convince a viewer that the runtime of `numOnes` is what you claimed it was.]

(C) How much time do you think it would take to run `numOnes` on an input of size $n = 10^{15}$?
[We are expecting: Your answer (in whichever unit of time makes the most sense) with a brief justification, that references the runtime data you generated in part (B). You don't need to do any fancy statistics, just a reasonable back-of-the-envelope calculation.]

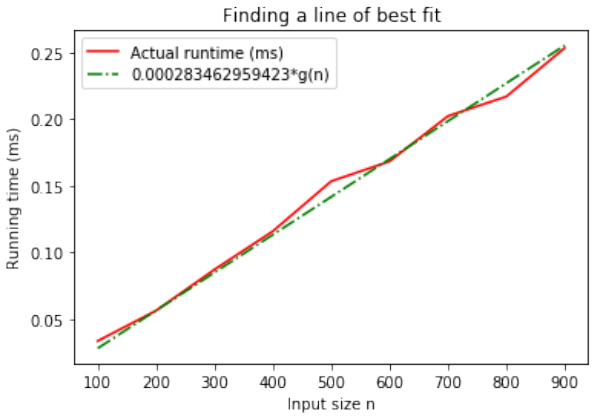
SOLUTION:

(A) The running time of `numOnes( lst )` on an input list of size n is $O(n)$. This is because the for loop goes through n iterations, and in each iteration it does a constant number of operations: it tests if $x == 1$, and then it might increment a counter. Thus, the running time is $O(n)$.

(B) See the picture below, where we have chosen $c = 0.00035$ and $n_0 = 100$. It seems in the picture that for all $n > n_0$ (that is, to the right of the yellow vertical line), the green dashed curve lies above the solid red curve, meaning that $cg(n) \geq T(n)$, where $g(n) = n$ and $T(n)$ is the running time of `numOnes` on an input of size n.



(C) Now that we've convinced ourselves through both parts (A) and (C) that the runtime is roughly linear, we can roughly model $T(n)$ (the running time of `numOnes` as $T(n) = a \cdot n$ for some slope a. In order to predict $T(10^{15})$ (without actually running it...) we should estimate a. For this problem, eyeballing it is fine for full credit. However, we chose to solve a least squares problem to find the *best* slope, which in our case turned out to be $a \approx 0.00028$. (Of course, this will be different on different computers, and even different runs on the same computer!) Projecting out, we find that the number of milliseconds that this would take is $10^{15} \cdot a$, which works out to about 283 billion milliseconds, or 3280 days, or about 9 years. Good thing we didn't try it!



3. **(4 pt.)** Using the definition of big-Oh, formally prove the following statements.

   (a) $2\sqrt{n} + 6 = O(\sqrt{n})$ (Note that you gave a "proof-by-picture" of this in Exercise 1).

   (b) $n^2 = \Omega(n)$

   (c) $\log_2(n) = \Theta(\ln(n))$

   (d) 4^n is **not** $O(2^n)$.

   [We are expecting: For each part, a rigorous (but short) proof, using the definition of $O(), \Omega()$, and $\Theta()$.]

   SOLUTION:

   (a) Choose $c = 4$ and $n_0 = 10$. We need to show that for all $n \geq n_0$, we have

   $$2\sqrt{n} + 6 \leq 4\sqrt{n}.$$

   Solving the above equation for n, it is equivalent to

   $$n \geq 9.$$

   This is certainly true for all $n \geq 10$, so we are done.
   Note: the solution above is written in a way that makes it clear how we got the answer. Another correct solution would be to go the other way: Choose $c = 4$ and $n_0 = 10$. Then it is true that for all $n \geq n_0$, we have

   $$\begin{aligned} n &\geq 9 \\ \sqrt{(n)} &\geq 3 \\ 2\sqrt{n} &\geq 6 \\ 4\sqrt{n} &\geq 2\sqrt{n} + 6 \\ c\sqrt{n} &\geq 2\sqrt{n} + 6. \end{aligned}$$

   Thus, for all $n \geq n_0$, $2\sqrt{n} + 6 \leq c\sqrt{n}$, hence $2\sqrt{n} + 6 = O(\sqrt{n})$.

   (b) Choose $n_0 = c = 1$. We need to show that for all $n \geq n_0$, we have

   $$n \leq n^2.$$

   Cancelling an n from each side (which we may do because $n > 0$, we see that this is equivalent to $n \geq 1$, which is true for all $n \geq n_0$.

(c) First, we note that $\log_2(n) = \ln(n)/\ln(2)$. We will show both $\log_2(n) = O(\ln(n))$ and $\log_2(n) = \Omega(\ln(n))$.
To show that $\log_2(n) = O(\ln(n))$, we choose $c = 1/\ln(2)$, and $n_0 = 1$. Then we have, for all $n \geq 1$,

$$\log_2(n) = \ln(n)/\ln(2) = c \cdot \ln(n),$$

so in particular, $\log_2(n) \leq c \cdot \ln(n)$, and this establishes that $\log_2(n) = O(\ln(n))$.
To show that $\log_2(n) = \Omega(\ln(n))$, we can do exactly the same thing, but note that the equality above also means that

$$\log_2(n) \geq c \cdot \ln(n)$$

for all $n \geq n_0$, and hence $\log_2(n) = \Omega(\ln(n))$.
We conclude that $\log_2(n) = \Theta(\ln(n))$.

(d) Suppose toward a contradiction that 4^n is $O(2^n)$. Then there is some c, n_0 so that

$$4^n \leq c2^n \quad \forall n \geq n_0.$$

Taking logarithms of both sides,

$$n \log(4) \leq \log(c) + n \quad \forall n \geq n_0.$$

Since $\log(4) = 2$ (the log is base 2, as with all logarithms in this course), this is the same as

$$n \leq \log(c) \forall n \geq n_0.$$

Choosing $n = n_0 + |\log(c)| + 1$ gives a contradiction, because this is an n which is both $\geq n_0$ and also $\geq \log(c)$.

Problems

You may talk with your fellow CS161-ers about the problems. However:

- Try the problems on your own *before* collaborating.
- Write up your answers yourself, in your own words. You should never share your typed-up solutions with your collaborators.
- If you collaborated, list the names of the students you collaborated with at the beginning of each problem.

---

1. **(4 pt.)** In class we discussed Karatsuba's algorithm for n-digit integers written in base 10. That is, for an integer x, we wrote $x = \sum_{i=0}^{n-1} x_i 10^i$, for $x_i \in \{0, \ldots, 9\}$. But we can also consider an n-bit integer y written in base 2: $y = \sum_{i=0}^{n-1} y_i 2^i$, for $y_i \in \{0, 1\}$. Or we can think about an n-hexadecimal integer z written in base 16: $z = \sum_{i=0}^{n-1} z_i 16^i$, for $z_i \in \{0, \ldots, 15\}$.[2]

   Your friend has come up with the following argument that integer multiplication can be done in $O(1)$ time. The argument has three parts:

   (a) Whatever base we choose to write the numbers out in, Karatsuba's algorithm correctly finds the product of those numbers. For example, if we wanted to multiply the numbers 11010011 and 01011010 (which are written in binary), we could do that by recursively performing three multiplications involving the numbers $1101, 0011, 0101$, and 1010.

   (b) For a given number x, the length of x's base-b representation is decreasing as b increases. For example, the same number $x = 1024$ (base 10) can be written as
   - 1000000000 base 2 (10 bits)
   - 1024 base 10 (4 digits)
   - 400 base 16 (3 hexits)

   (c) Suppose we want to multiply two numbers x and y. Part (b) means that there's some large-enough b so that the base-b representations of x and y have length $n = O(1)$. Then we run Karatsuba's algorithm in this base (which works by part (a)), and it takes time $O(n^{1.6}) = O(1)$ because $n = O(1)$. Therefore we can multiply any two integers in time $O(1)$.

   Unfortunately (from the perspective of fast integer multiplication) your friend's argument is flawed in at least one place. Which of their steps are faulty and why?

   [We are expecting: For each of (a), (b), and (c), either assert that your friend's logic is correct, or give a brief argument about why it is wrong. You do not need to give a formal proof in either direction.]

   SOLUTION:

   (a) This logic is correct. We can implement Karatsuba's algorithm no matter what the base is, by writing length-n numbers x and y as

   $$x = x_1 \cdot b^{n/2} + x_0$$

   and

   $$y = y_1 \cdot b^{n/2} + y_0.$$

[2] You may be used to representing number in hex on a computer, where it doesn't use symbols $0, \ldots, 15$ but rather $0, \ldots, 9$, along with the symbols A, B, C, D, E, F. In fact, this is the same thing, we just read "A" as 10, "B" as 11, and so on. So in hex, $1AF = 1 \cdot 16^2 + 10 \cdot 16^1 + 15 \cdot 16^0 = 431$ base 10.

Then we have

$$xy = x_1 y_1 b^n + (x_1 y_0 + x_0 y_1) b^{n/2} + x_0 y_0.$$

If we are storing our elements as vectors of numbers in $\{0, \ldots, b-1\}$, then multiplying by b^n is just shifting this representation to the left by n, so we can do that very quickly. The rest of Karatsuba's algorithm follows just as before.

(b) This logic is correct too. In general, we can write a number x in $n_b = \log_b(x)$ symbols in base b. Since n_b is equal to $\lfloor \log(x)/\log(b) \rfloor + 1$ (where both logs are base 2), then as b increases, n_b (weakly) decreases.

(c) This logic is the problematic part. The problem is that we chose b to depend on the original numbers x and y. This means that even though we do $O(1)$ *operations on the integers in* $\{0, \ldots, b-1\}$, the time it takes to do those operations is not $O(1)$ anymore – it depends on the size of the input. To take this to an extreme, if we work base $b = \max\{x, y\}$, then Karatsuba's algorithm doesn't do anything at all: the base case is exactly the same as the original problem!

2. **(10 pt.)** On an island, there are trustworthy toads and tricky toads. The trustworthy toads always tell the truth; the tricky toads may lie or may tell the truth. The toads themselves can tell who is tricky and who is trustworthy, but an outsider can't tell the difference: they all just look like toads.


You arrive on this island, and are tasked with finding the trustworthy toads. You are allowed to pair up the toads and have them evaluate each other. For example, if Tiffany the Toad and Tomás the Toad are both Trustworthy Toads, then they will both say that the other is trustworthy. But if Tiffany the Toad is a Trustworthy Toad and Tyrannus the Toad is a Tricky Toad, then Tiffany will call Tyrannus out as tricky, but Tyrannus may say either that Tiffany is tricky or that she is trustworthy. We will refer to one of these interactions as a "toad-to-toad comparison." The outcomes of comparing toads A and B are as follows:

Toad A	Toad B	A says (about B)	B says (about A)
Trustworthy	Trustworthy	Trustworthy	Trustworthy
Trustworthy	Tricky	Tricky	Either
Tricky	Trustworthy	Either	Tricky
Tricky	Tricky	Either	Either

Suppose that there are n toads on the island, and that there are strictly more than $n/2$ trustworthy toads.

In this problem, you will develop an algorithm to find all of the trustworthy toads, that only uses $O(n)$ toad-to-toad comparisons. Before you start this problem, think about how you might do this—hopefully it's not at all obvious! Along the way, you will also practice some of the skills that we've seen in Week 1. You will design two algorithms, formally prove that one is correct using a proof by induction, and you will formally analyze the running time of a recursive algorithm.

(a) **(1 pt.)** Give a straightforward algorithm that uses $O(n^2)$ toad-to-toad comparisons and identifies all of the trustworthy toads.

[We are expecting: A description of the procedure (either in pseudocode or very clear English), with a brief explanation of what it is doing and why it works.]

(b) **(3 pt.)*** Now let's start designing an improved algorithm. The following procedure will be a building block in our algorithm—make sure you read the requirements carefully!

Suppose that n is even. Show that, using only $n/2$ toad-to-toad comparisons, you can reduce the problem to the same problem with less than half the size. That is, give a procedure that does the following:

- **Input:** A population of n toads, where n is even, so that there are strictly more than $n/2$ trustworthy toads in the population.
- **Output:** A population of m toads, for $0 < m \leq n/2$, so that there are strictly more than $m/2$ trustworthy toads in the population.
- **Constraint:** The number of toad-to-toad comparisons is no more than $n/2$.

[We are expecting: A description of this procedure (either in pseudocode or very clear English), and rigorous argument that it satisfies the Input, Output, and Constraint requirements above.]

*This is the trickiest part of the problem set! You may have to think a while.

(c) **(1 bonus pt.) [This problem is NOT REQUIRED, but you may assume it for future parts.]** Extend your argument for odd n. That is, given a procedure that does the following:

- **Input:** A population of n toads, where n is odd, so that there are strictly more than $n/2$ trustworthy toads in the population.
- **Output:** A population of m toads, for $0 < m \leq \lceil n/2 \rceil$, so that there are strictly more than $m/2$ trustworthy toads in the population.
- **Constraint:** The number of toad-to-toad comparisons is no more than $\lfloor n/2 \rfloor$.

($\star$) *For all of the following parts, you may assume that the procedures in parts (b) and (c) exist even if you have not done those parts.*

(d) **(1 pt.)** Using the procedures from parts (b) and (c), design a recursive algorithm that uses $O(n)$ toad-to-toad comparisons and finds a *single* trustworthy toad.

[We are expecting: A description of the procedure (either in pseudocode or very clear English).]

(e) **(2 pt.)** Prove formally, using induction, that your answer to part (d) is correct.

[We are expecting: A formal argument by induction. Make sure you explicitly state the inductive hypothesis, base case, inductive step, and conclusion.]

(f) **(2 pt.)** Prove that the running time of your procedure in part (d) uses $O(n)$ toad-to-toad comparisons.

[We are expecting: A formal argument. Note: do this argument "from scratch," do not use the Master Theorem.]

(g) **(1 pt.)** Give a procedure to find *all* trustworthy toads using $O(n)$ toad-to-toad comparisons.

[We are expecting: An informal description of the procedure.]

SOLUTION:

(a) Consider the following algorithm:

```
trustworthyToad( toads ):
    for each toad T in the set of toads:
       score = 0
       for each toad T' other than T:
            compare(T,T')
            if T' says T is trustworthy:
                 score += 1
       if score >= n/2:
           return T
```

That is, we compare each toad with all other toads. If a majority of the remaining toads declares that this toad is trustworthy, the declare it to be a trustworthy toad.
To see that this works, consider two cases. In the first case, toad T is trustworthy, and there are at least $n/2$ trustworthy toads left in the population, since there were strictly more than $n/2$ before we removed T for testing. They will all declare toad T trustworthy, and we will also declare toad T trustworthy.
In the second case, toad T is tricky. Then there are at least $n/2$ trustworthy toads in the population, and at most $n/2 - 1$ tricky toads. So even if all the tricky toads vote that T is trustworthy, the coalition of $\geq n/2$ trustworthy toads will still win the vote, and we will not return T as trustworthy.

(b) Our procedure is as follows:

```
reduceByHalfForEvenN( toads ):
    Divide the n toads into n/2 pairs, {T0, T1}, {T2, T3}, ..., {T(n-2), T(n-1)}
    Ask each pair to evaluate each other
```

```
Let S be the set of pairs where both toads said "Trustworthy"
Initialize subToads = {}
for {Ti, T(i+1)} in S:
     add Ti to subToads
return subToads
```

That is, we pair up all the toads, throw out all the pairs that don't mutually declare each other trustworthy, and then select one toad from each remaining pair.

First, we observe that `subToads` contains no more than $n/2$ toads. Moreover `subToads` is not empty. This is true because, since there is a strict majority of trustworthy toads, at least one trustworthy toad must be paired with another trustworthy toad. Thus, `subToads` will contain one of this pair, and so it contains at least one toad.

Now that we've shown that the requirement on the size is satisfied, we need to show that more than half of the remaining toads are trustworthy.

Suppose that there are m pairs, $S = \{P_1, \ldots, P_m\}$ so that each toad in the pair said that their partner was trustworthy. And suppose that there were $n/2 - m$ pairs, $T = \{Q_1, \ldots, Q_{n/2-m}\}$ so that the toads said different things, or both said that their partner was tricky. Then we know the following things:

- For each $P_i \in S$, either both toads are trustworthy or both are tricky. Otherwise, the trustworthy one would have called out the tricky one.
- For each pair $Q_i \in T$, either both were tricky or one was tricky and one was trustworthy.

We claim that there are strictly more than $m/2$ (Trustworthy, Trustworthy) pairs in S. Otherwise, the number of Trustworthy Toads in the whole population would be

$$\#\text{ trustworthy toads } \leq 2\cdot \#(\text{Trustworthy, Trustworthy) pairs in S } + |T| \leq 2\cdot\frac{m}{2} + (n/2 - m) = n/2.$$

This is a contradiction, because we knew that there were strictly more than $n/2$ trustworthy toads to begin with.

Thus, when we choose one toad from each pair in S, we choose strictly more than $m/2$ trustworthy toads. This completes the argument.

(c) **[Bonus]** We'll follow the same idea as above; but now we can't pair up all the toads. The solution is to either include or not include the last toad, depending on the parity of the number of pairs we do get.

```
reduceByHalfForOddN( toads )
    oddToadOut = toads[0] # pick some special toad
    evenToads = all the toads except oddToadOut
    subToads = reduceByHalfForEvenN( evenToads )
    if len(subToads) is even:
        add oddToadOut to subToads
    return subToads
```

To see why this works, we analyze some cases.

- **Case 1**. `subToads` has an even number of elements.
  - **Case 1a**. There are strictly more trustworthy toads in `subToads` than there are tricky toads. Then there are at least two more trustworthy toads, since there's an even number. Then even if the oddToadOut is tricky, the result still has a strict majority of trustworthy toads.
  - **Case 1b**. There are the same number of trustworthy toads in `subToads` as tricky toads. Then the oddToadOut must be trustworthy, or there wouldn't have been a strict majority to begin with (this uses how our answer to part (b) works). So adding oddToadOut to the least creates a strict majority of trustworthy toads.

- **Case 1c**. There are more tricky toads than trustworthy toads in `subToads`. This can't happen or there wouldn't have been a strict majority to begin with.

- **Case 2**. `subToads` has an odd number of elements.
  - **Case 1a**. There are strictly more trustworthy toads in `subToads` than there are tricky toads. Then since we don't modify `subToads`, we win.
  - **Case 1b**. There are the same number of trustworthy toads in `subToads` as tricky toads. This can't happen since there are an odd number of toads in `subToads`.
  - **Case 1c**. There are more tricky toads than trustworthy toads in `subToads`. This can't happen or there wouldn't have been a strict majority to begin with.

(d) Finally, our algorithm is as follows:

```
findTrustworthyToad( toads ):
    if n == 1:
        return the only toad in the population.
    if n is even:
       return findTrustWorthyToad( procedureFromPartB( toads ) )
    if n is odd:
       return findTrustWorthyToad( procedureFromPartC( toads ) )
```

(e) To prove that our algorithm is correct, we will maintain the recursive invariant that each call to `findTrustworthyToad` is correct. More precisely:

- **Inductive Hypothesis.** For all $m \leq n$, if `toads` contains m toads with $> m/2$ trustworthy ones, `findTrustworthyToad( toads )` will return a single trustworthy toad in `toads`.
- **Base case.** When $n = 1$, then for all $m \leq 1$ (namely, $m = 1$, since we will never return a set of size 0 by parts (b) and (c)), we need to show that `findTrustworthyToads` will return a trustworthy toad. This is true because if $m = 1$, then this single toad must be trustworthy (since a strict majority is), and we return it.
- **Inductive step.** We need to show that if the inductive hypothesis holds for n, then it holds for $n+1$. This is exactly the statement that we proved in part (b) (for even n) and part (c) (for odd n).
- **Conclusion.** By induction, we conclude that the inductive hypothesis holds for all $n \geq 1$. In particular, this means that if we run this procedure when we arrive at the island with n toads, the procedure will find a trustworthy toad.

(f) Next, we address the number of comparisons. Suppose that our first problem size is $n_0 = n$, our second is $n_1 \leq \lceil n_0/2 \rceil \leq n_0/2 + 1$, our third is $n_2 \leq \lceil n_1/2 \rceil \leq n_1/2 + 1$ and so on:

$$n_i \leq n_{i-1}/2 + 1.$$

Iterating this, we find

$$n_i \leq \frac{n_{i-1}}{2} + 1 \leq \frac{1}{2}\left(\frac{n_{i-2}}{2} + 1\right) + 1 = \frac{n_{i-2}}{4} + \frac{1}{2} + 1$$

and if we keep repeating this, we find that, for all i,

$$n_i \leq \frac{n_0}{2^i} + \sum_{j=0}^{i-1} 2^{-i} \leq \frac{n}{2^i} + 1.$$

Moreover, at each level i, we do at most $n_i/2$ comparisons. Thus, the total number of

comparisons is

$$\sum_{i=0}^{k} \text{number of comparisons at level } i \leq \sum_{i=0}^{k} n_i/2$$
$$\leq \frac{1}{2}\sum_{i=0}^{k}(n/2^i + 1) \leq \frac{n}{2}\sum_{i=0}^{k}\frac{1}{2^i} + \frac{k}{2}.$$

Now, $k \leq \log(n)$ (since we can't divide the input in half more than $\log(n)$ times), and we also have $\sum_{i=0}^{\infty} 1/2^i = 2$, so we the sum is at most

$$\text{total number of comparisons} \leq n + \frac{\log(n)}{2} = O(n).$$

NOTE: You could get full credit on this problem even if you did not pay attention to floors and ceilings.

(g) First, use $O(n)$ comparisons to find a single trustworthy toad. Then use n more comparisons to have that trustworthy toad tell you which of the remaining toads are tricky and which are trustworthy.