Algorithmic Combinatorics
Shagnik Das

Wi 2016–17
Tibor Szabó

Some notes on an exercise

The Exercise

Sheet 1, Exercise 2 Consider the following game. I think of an integer x between 1 and n, and your job is to try and determine x. You are allowed to ask questions of the form "Is $x < y$?" or "Is $x > y$?" for any y.

(a) Show that you can find x with only $\lceil \log_2 n \rceil$ questions, and that this is best possible.

To make your job slightly harder, I am now allowed to lie to you at most k times, for some constant k.

(b) How many questions do you now need to determine x? Provide the best lower and upper bounds that you can find.

A solution

Part (a)

The upper bound To show that we can find x within $\lceil \log_2 n \rceil$ questions, we use a binary search algorithm.[1] Suppose after asking t questions we have determined that $x \in I_t$ for some interval I_t, starting with $I_0 = [1, n]$. If I_t only has one integer in it, then x must be that integer, and we are done.

Otherwise take y to be the midpoint of I_t and ask, "is $x < y$?" This divides I_t into a left and right interval of equal length, so no matter what the answer is, we can find some I_{t+1} of length exactly half of I_t. Since we start with an interval of length $n - 1$, within $\lceil \log_2 n \rceil$ steps[2] we must have an interval of length strictly less than 1, which can contain at most one integer, at which point we can terminate the algorithm.

[1] This is very similar to what we did with the binary insertion sorting algorithm

[2] We need to have *strictly* more than $\log_2(n-1)$ questions to guarantee the final interval has length smaller than 1 (and hence cannot contain two integers). $\lceil \log_2 n \rceil$ is the smallest integer strictly larger than $\log_2(n-1)$.

The lower bound We now show that you cannot do better. Suppose we had an algorithm that asks at most q questions. By asking redundant questions if necessary, we can assume that every run of the algorithm asks exactly q questions. Since each question is a yes/no question with two possible answers, there are at most 2^q possible sequences of answers.

If the algorithm correctly uniquely identifies the chosen number x, then the algorithm must be a surjective function from the set of possible sequences of answers to the n possible numbers. Hence we require $2^q \geq n$, or equivalently, $q \geq \lceil \log_2 n \rceil$.

Part (b)

We now play against an opponent who can tell a limited number (k) of lies[3], and investigate how this affects the number of questions we need to ask.

Some first bounds

A lower bound Note that our opponent could simply decide never to lie, in which case we would be playing the same game as (a). Hence the lower bound from there still holds in this more general (i.e. difficult) game, so we still need at least $\lceil \log_2 n \rceil$ questions.

An upper bound For an upper bound, notice that if we could force our opponent to reveal a truthful answer, then we could follow the same binary search algorithm as in part (a), forcing a truthful answer at every step.

To force a truthful answer, we simply ask the same question repeatedly. If we see the same answer $k+1$ times, then it must be the truth, since the opponent is not allowed to lie that many times. Hence we follow the same strategy as in (a), except we ask every question repeatedly until we see the same (truthful) answer $k+1$ times.

This means in total we will see $(k+1)\lceil \log_2 n \rceil$ truthful answers. Along the way our opponent could lie at most k times, so we ask a total of at most $(k+1)\lceil \log_2 n \rceil + k$ questions.[4]

An asymptotic answer

Intuitively, one feels that one should be able to do better than simply repeating each question several times — surely there must be some clever strategy that lets us gain information with each question? However, the repetition strategy is quite natural. There is, in fact, a subtle way to use repetition to get a sharper result.

Rather than repeat every one of the $u = \lceil \log_2 n \rceil$ questions until we get the same answer $k+1$ times, we shall break the sequence of u questions down into blocks of r consecutive questions. We shall ask all the questions in a block assuming that the answers we receive

[3] Is it not curious how "lie-ability" and "liability" sound so similar?

[4] The astute reader might note that if our opponent lies early, we will detect this, and then in later steps we do not need to get $k+1$ repeated answers, since there will be fewer lies available. However, if the opponent saved all the lies for the last round of questions, then there would be no such savings.

are correct. When we reach the end of the block, we will be operating under the assumption $x \in I_t = [a_t, b_t]$. We will now check whether this is true by repeatedly querying the endpoints $k+1$ times: "$x > a_t - 1$?" and "$x < b_t + 1$?"[5] If each of these questions receives $k+1$ positive answers, then we know that we are working in the right subinterval, and resume the bisection algorithm with the next block of r questions.

If we instead get a negative response, then there must have been a lie somewhere in the block. We take a conservative point of view and assume that this whole block is unreliable, and start again from the beginning of the block.

Each successful block takes $r + 2(k+1)$ questions — r for the block itself, and $2(k+1)$ for the verification at the end. We need $\frac{u}{r}$ successful blocks, and can have to repeat at most k blocks (one per lie). Hence the total number of questions asked is at most

$$(r + 2(k+1)) \left(\frac{u}{r} + k\right) = u + rk + \frac{2(k+1)u}{r} + 2(k+1)k.$$

To minimise this upper bound, we choose $r \approx \sqrt{2u}$ to get an upper bound of

$$u + (2k+1)\sqrt{2u} + 2(k+1)k \sim \log_2 n + (2k+1)\sqrt{2\log_2 n} + 2(k+1)k.$$

Note that for k fixed, this asymptotically matches the lower bound of $\log_2 n$, avoiding the multiplicative factor of k we had earlier.

A much better upper bound

This still leaves a gap between the bounds, and in particular does not give the right dependence on the number of lies k. A more involved argument can in fact give a very tight bound. Luckily for us, we had Ander to guide the way!

Bookkeeping If we recall the "playing against the Devil" paradigm, we thought of the Devil as not fixing a number to begin with, but keeping track of which answers are still available to him on the fly. When we are forming a strategy against a lying opponent, it makes sense for us to take a similar approach. As we receive answers to our questions, we keep track of which numbers are still viable, and use this information to choose the next question we ask.

Every time we ask a question, we divide the set of n possible numbers into two subsets. Whatever the opponent answers, one of those subsets gets a 'No', which means that if one of those numbers was in fact the correct one, then a lie has been told. Hence, for $1 \leq i \leq n$, after having asked q questions, we can let $L_q(i)$ denote the number of lies that would have been told if i was in fact the correct number. Note that we have $L_0(i) = 0$ for all i, and if $L_q(i) \geq k+1$, then we can safely rule i out.

[5] The old adage in practice: "trust but verify."

Objective Now that we have a way to keep track of the potential answers, what should we be aiming for? As we observed above, if at some point $L_q(i) \geq k+1$, then we can rule i out, as there would have been too many lies for i to be the correct number. This is in fact an if-and-only-if statement, since if $L_q(i) \leq k$, then at most k lies would have been used up at this stage, and the opponent could continue to answer truthfully for i as the correct number.

We are done precisely when there is only one viable candidate left, so $L_q(i) \geq k+1$ for all but one choice of i. Hence the problem is equivalent to minimising the number of questions q needed until there is a unique i with $L_q(i) \leq k$.

Weights Now that we have restated our objective, we have to use it to find an algorithm for determining which question to ask next. Intuitively, you might think that since we want to reduce the number of options i with $L_q(i) \leq k$, we should split the interval to halve the number of such i on each side.[6] This way, no matter what our opponent answers, we will always increase $L_q(i)$ by 1 for at least half of the remaining viable i.

The problem with this approach is that it does not differentiate between the precise values taken by $L_q(i)$ — a number i with $L_q(i) = 0$ is treated the same as a number i' with $L_q(i') = k$. However, if our opponent lies about i' even once more, then we can safely rule it out, while he can lie about i many more times. Thus i is in some sense more dangerous for us, and this should somehow be reflected in our strategy. Without taking the actual values of $L_q(i)$ into account, we are not gaining anything from our bookkeeping.[7]

We would thus like to somehow use the individual values $L_q(i)$ in a finer way. Note that the number of lies remaining for the number i before we can rule it out is given by $\max(k+1-L_q(i), 0)$. One might think to instead sum up these number of remaining lies, and then split the interval so that the sums on either side are as equal as possible. The reasoning behind this is that those dangerous numbers i with many lies remaining contribute more to the sum, and hence we would tend to decrease their remaining-lie count more, moving towards having fewer numbers with lies left (that could therefore be legitimate answers).

The issue with this method is that it is hard to keep track of how the function we are summing, $\max(k+1-L_q(i), 0)$, changes. After asking a question, we know that the opponent lied about one part of the interval that covers half of the sum $\sum_i \max(k+1-L_q(i), 0)$. However, this sum now decreases by 1 for every viable number i in the lied-about part of the interval, but we have no way of knowing how many numbers this actually is — it could be a small number of numbers, each with a lot of lies left, or a large number of numbers with few lies left. Since we cannot closely follow how this sum decreases over time, it will be difficult to analyse what questions are asked and to know when we are done.

A neat trick to get around these difficulties is to use exponential weights. For each number i, after q questions define the weight $w_q(i) = 2^{k-L_q(i)}$. Note that as $L_q(i)$ increases, $w_q(i)$ decreases, so the dangerous numbers with a lot of lies remaining get the highest weights.

[6] A moment's thought reveals that this is always possible. We can ignore any elements with $L_q(i) \geq k+1$, and then take y to be the median of whatever numbers remain.

[7] In fact, closer inspection reveals that this approach would be exactly the same as the first repetition algorithm from part (b) above, which we are trying to beat.

This allows us to tailor our questions to take care of the more dangerous numbers, for which our opponent has the most room to manoeuvre.

However, regardless of what $L_q(i)$ currently is, if the opponent tells a lie about i in the next question, then $L_{q+1}(i) = L_q(i) + 1$, and so $w_{q+1}(i) = \frac{1}{2}w_q(i)$. Hence lies decrease the total weight in a *linear* fashion, which makes it easy to keep track of how the weight evolves as our algorithm proceeds. This motivates our use of $w_q(i) = 2^{k-L_q(i)}$ in what follows.

A weighted objective So how should we use these weights? Observe that a number i could possibly be the answer if and only if $L_q(i) \leq k$, which is if and only if $w_q(i) \geq 1$.[8] Hence our goal is to reduce the weights of the numbers until the second largest weight satisfies $w_q(i) < 1$; at this point, we know that the number with the largest weight (this weight must be at least 1) must be the correct answer.

However, recall that the nice thing about these weights is that they behave linearly with respect to lies, while the maximum (or second-largest) of a sequence is not a nice linear function. Hence, rather than looking at the second-largest of the weights, it will be more convenient to consider the sum of the weights (excluding the largest). Since the weights are always positive, we know that the sum is an upper bound on each summand. Therefore if the sum of all but the largest weight is less than 1, the second-largest weight must be less than 1, and we are done.

To this end, after q questions, let $m_q = \max_i w_q(i)$ represent the largest remaining weight, and let $t_q = \sum_{i=1}^{n} w_q(i)$ represent the total sum of weights. The quantity we are looking to bound is then $f_q = t_q - m_q$. If we can guarantee $f_q < 1$ for some q, then the only viable option remaining is the number with the highest weight.

Choosing the right questions With all these preliminaries out of the way, we can now finally describe the algorithm for determining our opponent's number. Our goal will be to ask questions that cause the quantity $f_q = t_q - m_q$ to decrease rapidly, until it will eventually be smaller than 1, at which point we can stop. Note that at the start, each number has k lies remaining, so $w_0(i) = 2^k$ for all i, and $f_0 = (n-1)2^k$.

Another key observation is that the weights $w_q(i)$ are non-increasing in q — as we ask more questions, the number of lies remaining, and hence the weights, cannot increase. In particular, this implies that the quantity f_q is also non-increasing, so $f_{q+1} \leq f_q$ for all q. To see this, first suppose $m_{q+1} = m_q$, so that the maximum weight doesn't drop after the $(q+1)$st question. Then $f_{q+1} - f_q = t_{q+1} - t_q = \sum_i (w_{q+1}(i) - w_q(i)) \leq 0$, since the weights are non-increasing. On the other hand, if the maximum does drop, we must have $m_{q+1} = \frac{1}{2}m_q$, since the weights can drop by at most a factor of two. Let i^* be a number achieving the maximum weight after q questions (i.e. $w_q(i^*) = m_q$). We then must have $w_{q+1}(i^*) = m_{q+1}(= \frac{1}{2}m_q)$. Thus $f_q = t_q - m_q = t_q - w_q(i^*) = \sum_{i \neq i^*} w_q(i)$, and similarly

[8] You might wonder why we assign fractional weights to numbers with $L_q(i) \geq k+1$; perhaps it seems more natural to give them weight 0 since they can be ruled out completely. This may be true, but it is convenient to keep the fact that lying always decreases the weight of a number by a factor of $\frac{1}{2}$. Besides, these fractional parts are quite small, so they do not have a great effect on the total system.

$f_{q+1} = \sum_{i \neq i^*} w_{q+1}(i)$. Using the non-increasing nature of the weights, we then again have $f_{q+1} \leq f_q$.

This shows that the objective function f_q cannot increase. The next, crucial, claim shows that in fact it often decreases sharply.

Claim 1. *For every $q \geq 0$, we can find a number y_{q+1} such that after receiving the answer to the question "is $x < y_{q+1}$", one of the following holds:*

(i) $m_{q+1} = \frac{1}{2} m_q$, *or*

(ii) $f_{q+1} \leq \frac{3}{4} f_q$.

Our algorithm is then simply to use the claim to find which questions to ask at every step. Before we prove the claim, though, let us analyse what this algorithm will give us.

Proposition 2. *If the opponent is allowed to lie at most k times, we can determine the correct number in at most $\log_{4/3}\left((n-1)2^k\right) + k + 1$ questions.*

Proof of Proposition 2. Suppose that our algorithm has not terminated after q questions. Observe that we must have $m_q \geq 1$, since any possible number has weight at least 1. In particular, since we start with $m_0 = 2^k$, this means that case (i) in Claim 1 occurs for at most k of the q questions.

Hence case (ii) occurs at least $q - k$ times, so we have $f_q \leq \left(\frac{3}{4}\right)^{q-k} f_0$. Recall that $f_0 = (n-1)2^k$, and we must have $f_q \geq 1$, as otherwise there is a unique viable answer, and we are done. Hence $1 \leq \left(\frac{3}{4}\right)^{q-k} (n-1)2^k$, and solving for q we find $q \leq \log_{4/3}\left((n-1)2^k\right) + k$.

Thus after at most $\log_{4/3}\left((n-1)2^k\right) + k + 1$ questions, we must be done. □

To write this bound in more familiar terms, we observe that $\log_{4/3} 2 \approx 2.41$, and so the upper bound is approximately $2.41 \log n + 3.41k$. This is only a (reasonably small) constant factor larger than our lower bound of $\lceil \log_2 n \rceil$, as opposed to our previous upper bound of $(k+1)\lceil \log_2 n \rceil + k$, where we gained a multiplicative factor of k. Hence this shows that, provided k is small compared to $\log n$, we still need $\Theta(\log n)$ questions to determine the liar's number.

It remains to prove Claim 1, which we do now.

Proof of Claim 1. Our goal is to split the interval in half with respect to the weights $w_q(i)$. This way, no matter what answer the opponent gives us, he will lie for numbers corresponding to (roughly) half the weight. Since each lie cuts the weight in half, this means that in total, we lose a quarter of the weight, which will give rise to the $\frac{3}{4}$ ratio.

Formally, let y_{q+1} be the minimum integer such that $\sum_{i < y_{q+1}} w_q(i) \geq \frac{1}{2}(t_q - m_q)$. Note that by minimality of y_{q+1}, we have $\sum_{i < y_{q+1}-1} w_q(i) < \frac{1}{2}(t_q - m_q)$, and so

$$\sum_{i < y_{q+1}} w_q(i) = \sum_{i < y_{q+1}-1} w_q(i) + w_q(y_{q+1} - 1) < \frac{1}{2}(t_q - m_q) + w_q(y_{q+1} - 1) \leq \frac{1}{2}(t_q - m_q) + m_q,$$

where the last inequality follows because m_q is the maximum weight of a number. Hence we have $\sum_{i<y_{q+1}} w_q(i) < \frac{1}{2}(t_q + m_q)$, and so the complementary part satisfies

$$\sum_{i \geq y_{q+1}} w_q(i) = t_q - \sum_{i<y_{q+1}} w_q(i) > t_q - \frac{1}{2}(t_q + m_q) = \frac{1}{2}(t_q - m_q).$$

Hence the interval is split into two parts, each of weight at least $\frac{1}{2}(t_q - m_q)$. No matter what answer the opponent gives, he will be lying about one of these two intervals, and so we will lose half the weight of that interval. Thus we find that

$$t_{q+1} \leq t_q - \frac{1}{2} \min \left(\sum_{i<y_{q+1}} w_q(i), \sum_{i \geq y_{q+1}} w_q(i) \right) \leq t_q - \frac{1}{2} \cdot \frac{1}{2}(t_q - m_q) = \frac{3}{4} t_q + \frac{1}{4} m_q.$$

If we find that after the opponent's answer, the maximum weight has dropped, then (since the weights are all powers of two), we must have $m_{q+1} = \frac{1}{2} m_q$, and we are in case (i).

Otherwise the maximum stays the same, so $m_{q+1} = m_q$, and we have

$$f_{q+1} = t_{q+1} - m_{q+1} = t_{q+1} - m_q \leq \frac{3}{4} t_q + \frac{1}{4} m_q - m_q = \frac{3}{4}(t_q - m_q) = \frac{3}{4} f_q,$$

and so we are in case (ii). This completes the proof of the claim. □

Summary To quickly recap the ideas behind this much-improved upper bound, we wanted to keep track of which numbers had a lot of lies left, and try to force the opponent to lie for some of these numbers. We found that an exponential weight function gave us the right compromise between sensitivity to numbers with many lies remaining, and control over its evolution over time.

Our goal was to ensure that the second-largest weight was smaller than 1, which would leave at most one possible number for our opponent. Due to the linear nature of our weights, it made sense to look at the sum of all but the largest weight instead. Claim 1 then said that we could find a question which would either decrease the maximum weight, or decrease this sum by a factor of $\frac{3}{4}$. This then allowed us to prove Proposition 2, showing that the number of questions needed is at most $2.41 \log_2 n + 3.41k$, just a constant multiplicative factor larger than our lower bound (provided k is small compared to $\log n$).

A more precise answer

At this point there might be one thing[9] you do not quite understand: why did we choose 2 to be the base of our exponential weights? Hopefully our earlier discussion has convinced you that it is reasonable to take exponential weights, since these give us the nice linear behaviour we have taken advantage of.

[9] Or, indeed, more than one, but there is only one thing I wish to focus on at this point.

However, in case (ii) of Claim 1, we found that the objective function f_q only decreased by a factor of $\frac{3}{4}$ with every question, and this lead to the appearance of $\log_{4/3} n$ in our upper bound. If we compare to part (a), where lying was not permitted, we found that we could instead decrease the number of possible options by a factor of $\frac{1}{2}$ each time, and this gave us a $\log_2 n$ that matched our lower bound.

Now that we are comfortable with the method, we can try experimenting with different bases, in the hope of getting something closer to the lower bound. More generally, we could take $w_q(i) = \alpha^{k-L_q(i)}$ for some $\alpha > 1$. In this case we would have $f_0 = (n-1)\alpha^k$ instead, and every lie would decrease the weight of a number by a factor of $\frac{1}{\alpha}$.

If you chase through the previous argument,[10] you will find that in our claim, we either have

(i) $m_{q+1} = \frac{1}{\alpha} m_q$, or

(ii) $f_{q+1} \leq \frac{\alpha+1}{2\alpha} f_q$.

This is a good sign — note that now in case (ii), the objective function decreases by a factor of $\frac{\alpha+1}{2\alpha}$. As $\alpha \to \infty$, this ratio tends to $\frac{1}{2}$, which gets closer to what we had in part (a). However, as α grows, so too does the initial value $f_0 = (n-1)\alpha^k$, and so we will have to balance these conflicting quantities.

When we repeat the calculations from Proposition 2, the upper bound now becomes $\log_{2\alpha/(\alpha+1)} \left((n-1)\alpha^k\right) + k + 1$. We shall now do some asymptotic calculations to find the value of α that gives the best possible upper bound. Throughout this we shall assume that k is small compared to $\log_2 n$.[11]

We shall first wish to convert to binary logarithms. To this end, note that $\log_2 \frac{2\alpha}{\alpha+1} = 1 + \log_2 \frac{\alpha}{\alpha+1} = 1 + \frac{\ln \frac{\alpha}{\alpha+1}}{\ln 2} \geq 1 - \frac{1}{\alpha \ln 2}$, where we use the bound $\ln x \geq 1 - \frac{1}{x}$ for $x > 0$.

Hence we have

$$\log_{2\alpha/(\alpha+1)} \left((n-1)\alpha^k\right) = \frac{\log_2 \left((n-1)\alpha^k\right)}{\log_2 \frac{2\alpha}{\alpha+1}} \leq \frac{\log_2 \left(n\alpha^k\right)}{1 - \frac{1}{\alpha \ln 2}} \leq \left(1 + \frac{3}{\alpha}\right) \log_2 \left(n\alpha^k\right),$$

provided $\alpha > 3$, say. This means that our upper bound is at most

$$\left(1 + \frac{3}{\alpha}\right) \log_2 \left(n\alpha^k\right) + k + 1 = \log_2 n + \frac{3 \log_2 n}{\alpha} + k \left[\left(1 + \frac{3}{\alpha}\right) \log_2 \alpha + 1\right] + 1.$$

For the $\frac{\log_2 n}{\alpha \ln 2}$ to be a lower-order error term, we could take $\alpha = \log_2 n$ (or something of a comparable rate of growth). Since the k gets multiplied by $\log_2 \alpha$, it would not be in our interests to make α much bigger. This leaves us with the following final bound.

Theorem 3. *If the opponent is allowed to lie at most k times, where $k = o(\log n)$, we can determine the correct number in at most $\log_2 n + k (\log_2 \log_2 n + 2) + 2$ questions.*

[10] I strongly encourage you to do this!

[11] In the exercise, we assumed k was constant, but it could be allowed to grow slowly with n.

What about the lower bound?

Our lower bound came from part (a), where we did not allow any lies. Intuitively, we would expect that allowing the opponent to lie should make our lives a little harder.[12] Here we briefly sketch an argument that shows the dependence on k in Theorem 3 is essentially correct.[13]

Theorem 4. *If the opponent is allowed to lie at most k times, where $k = o(\log n)$, then we need at most $\log_2 n + k \log_2 \log_2 n + k$ questions to determine the correct number.*

Proof of Theorem 4. In order to prove a lower bound, we need to give a strategy for the opponent — how could he play to force us to ask a lot of questions?

Note that our questioning strategy from Theorem 3 is designed so that no matter what answer the opponent gives, the objective function is roughly halved. This means that against our strategy, the opponent could play arbitrarily, or even randomly, and we would still be done reasonably quickly.

Of course, the opponent does not know that we will necessarily play with this strategy. However, if the opponent does indeed play randomly, then we can analyse his performance independently of what strategy we actually play with (!), and can then hopefully show a good lower bound.

Thus the strategy of the random opponent is the following: for every question we ask, the opponent will answer "Yes" with probability $\frac{1}{2}$, and "No" with probability $\frac{1}{2}$, independently of all previous answers.

If this happens for q questions, we say that a number $i \in [n]$ survives if it receives at most k lies. Note that the number of lies $L_q(i) \sim \mathrm{Bin}\left(q, \frac{1}{2}\right)$ is binomially distributed. This is because the opponent doesn't even consider what the question is, and the outcome depends solely on the result of the fair coin toss.

Hence the probability that the ith number survives after q questions is

$$\mathbb{P}\left(\mathrm{Bin}\left(q, \frac{1}{2}\right) \leq k\right) = 2^{-q} \sum_{i=0}^{k} \binom{q}{i} \geq 2^{-q} \binom{q}{k} \geq \frac{q^k}{2^q k^k}.$$

By linearity of expectation, the expected number of surviving numbers after q questions is thus $\frac{nq^k}{2^q k^k}$. If this expectation is bigger than 1, then with positive probability there must be at least 2 surviving numbers, in which case our algorithm cannot terminate.

Of course, this is just with positive probability, but the probability could (a priori) be very small. However, if we had an algorithm that only needed q questions, we would *always* be done in q questions, and so the probability of having 2 numbers surviving would be 0.

Hence if $\frac{nq^k}{2^q k^k} > 1$, or $\frac{2^q}{q^k} < \frac{n}{k^k}$, then we would need at least $q + 1$ questions.

[12] Although our upper bound shows that it cannot make it too much harder — the k only appears in the main error term, and we would need to allow $\Omega(\log n / \log \log n)$ lies before it would affect the leading $\log_2 n$ term.

[13] This matches the upper bound up to an additive error of $k + 2$, which is remarkably close. In fact, with some more careful bounding, the error is really $o(k) + 2$.

Taking logarithms, this means any $q < \log_2 n - k\log_2 k + k\log_2 q$ will do. Since we certainly have $q < 2k\log_2 n$ (recall[14] our first upper bound on this problem), we can substitute this bound in on the right-hand side to get the desired lower bound of

$$\log_2 n - k\log_2 k + k\log_2(2k\log_2 n) = \log_2 n + k\log_2\log_2 n + k. \qquad \square$$

Historical remarks

This problem is a variant of what is known as Ulam's game, as it was first posed by Stanislas Ulam in his autobiography [2]. The asymptotic upper bound from pages two and three is due to M. A. Spencer, and a general solution to a very closely related problem was given by J. Spencer. [1].

References

[1] J. Spencer, *Ulam's searching game with a fixed number of lies*, Theoretical Computer Science **95** (1992), 307–321.

[2] S. M. Ulam, *Adventures of a mathematician*, Scribner, New York (1976), 281.

Epilogue

I hope that this exposition of Ander's argument for the improved upper bound has helped clarify the situation, and shed some light on the ideas behind the algorithm.[15] At the same time, I fear that this note may have caused some of you concern, and I wanted to make use of this space to allay any fears you may have.

The first thing that I want to say is that this is a pretty involved argument, and it is not one that you should expect to fully understand at first sight. I have attempted to make it more accessible in this note by seeking to explain why things are chosen the way they are, and why some alternatives might not work, but you still might have to read through it a few times before it makes much sense. We will encounter some of these ideas (in a more basic form) when we talk about derandomisation towards the end of the course, and then we will have some further intuition coming from the probabilistic side of things, so that might help make it clearer.

The calculations here are also quite technical, especially those on pages seven to nine. The calculations with the weights $w_q(i) = 2^{k-L_q(i)}$ should hopefully be a little easier to parse,

[14] From page 2, if you can recall that far!

[15] I like to imagine that you are in a warm, comfortable room, hot cup of cocoa at your side, enjoying reading this note in front of a non-Apple computer. You laugh to yourself as you appreciate the cleverness of these weights, pull up some scrap paper to check the arithmetic, let out a "tut tut" in disapproval as you find some mistakes in my calculations, and then send me an e-mail to let me know what to fix and how better to present the argument.

but since this argument seems to give a (very) tight bound, I thought it worth including the general bound with α, and the improved random lower bound. Those of you who took the Probabilistic Method course last semester are hopefully well-practised at such calculations, but I would expect that this asymptotic manipulation is new to many of you. We will have gentler introductions to working with asymptotics as they arise in the course.

Finally, you might be thinking to yourself, "do I really need to write eleven pages to get full points on a single homework exercise?" The answer is, of course, no! I would have been happy with answers similar to part (a) and the first bounds for part (b) on pages one and two. That said, I am overjoyed to have received this improved solution, and look forward to seeing many more of your breakthroughs in the weeks to come!