A Binary Guessing Game

Tom Davis
tomrdavis@earthlink.net
http://www.geometer.org/mathcircles
October 13, 2016

Abstract

We discuss a guessing game that has interesting features related to both mathematics and computer science and algorithms.

1 The Rules

Rules: One person chooses a secret n-bit pattern and the other guesses n-bit patterns until they know what the secret pattern is. (You could play by two sets of rules; namely, the second person has to guess the correct pattern, or the second person only has to guess until they know the secret pattern. We'll use this second interpretation, realizing that to convert it to the first situation would sometimes require one more "guess" which isn't really a guess, since the pattern has been completely identified.) The goal, of course, is to identify the pattern with as few guesses as possible.

For example, if $n = 1$, exactly one guess is required: if you guess 1 either all your bits are correct or all incorrect, and in either case you know the secret. For $n = 2$, you need two guesses (proof?).

2 Straight-Forward Strategies

Unless we say otherwise, we'll use $n = 8$ here for most of the examples.

Here's a straight-forward guessing strategy: First guess is $p_0 = 11111111$ and the result will be a number k that counts the number of 1's in the secret pattern. Then, successively guess $p_1 = 01111111$, $p_2 = 00111111$, $p_3 = 00011111$, and so on, up to $p_7 = 00000001$.

The first guess tells us the total number of 1's in the secret pattern, since only 1's will generate a hit. For each successive pattern, since it differs by one bit from the previous, has to yield a match number that is either one more or one less than the previous. If the match number goes down, we found a 0; if it goes up, we found a 1. We don't have to go to the last position, since by that point we will have found either $k - 1$ or k of the 1's, so we know the value of the last bit.

Let's look at using this sequence when the secret pattern is 11010101. Check to see that these are the match counts for patterns $p_0, p_1, \ldots, p_7$: $5, 4, 3, 4, 3, 4, 3, 4$, so there are 5 total ones (according to p_1) then the numbers go: down, down, up, down, up, down, up. Since "down" translates to 1, this corresponds to $1101010x$, but since we've identified only 4 of the 5 1's, the x has to be 1.

Note that if you're following this sequence and find the k^{th} 1, you can stop, since all the rest must be 0, so the guessing could end quite early if k is small. We'll call this strategy "Find the 1's."

So there's a completely mechanical way to guess the pattern with exactly n (or fewer) guesses (or up to $n + 1$, if you need to "guess" the known final result). Obviously, if you announce that this is your strategy (or if you just play it repeatedly), your opponent can always require you to take the maximum number of steps by picking any pattern that ends with a 1.

We can also consider two other versions of the game. In the first version, the guesser can modify his guesses depending on the answers he gets to previous guesses and in the second, he has to submit his list of choices beforehand. Obviously, we'd expect to get faster results in the first case.

Here's a method to win fairly quickly when the choices can be modified, depending on previous results:

1. Guess 00000000. The answer you get will eliminate a bunch of possibilities, so imagine starting with a list of all 256 possible 8-bit strings and crossing off all the impossible ones, depending on the answer to the 00000000 query.
2. Among the non-eliminated possibilities, choose the smallest (interpreting each pattern as an 8-bit number) pattern that's still possible, and use that as your guess. If you got it, you're done, otherwise, cross off all the ones that your answer made impossible, and repeat this step.

Let's call this strategy "Try a possible."

If we assume that you're playing against an opponent who is equally-likely to choose any of the 256 possibilities (a somewhat stupid strategy – maybe) then here are the average number of guesses required to find that pattern (how do you write efficient computer code to generate this table?):

n	Patterns	Count	Average	Maximum
1	2	2	1	1
2	4	6	1.5	2
3	8	18	2.25	3
4	16	46	2.875	4
5	32	114	3.5625	5
6	64	261	4.07812	6
7	128	610	4.76562	7
8	256	1348	5.26562	8
9	512	2953	5.76758	9
10	1024	6374	6.22461	10
11	2048	13690	6.68457	11
12	4096	29078	7.09912	12
13	8192	61565	7.51526	13
14	16384	129754	7.91956	14
15	32768	272415	8.31345	15
16	65536	570076	8.69867	16
17	131072	1188145	9.06483	17
18	262144	2466496	9.40894	18
19	524288	5108742	9.74415	19
20	1048576	10566057	10.0766	20
21	2097152	21818393	10.4038	21
22	4194304	44988068	10.726	22

Here's how to interpret the table. The first column is n, the number of bits in the pattern to be guessed. The "Patterns" column is just 2^n, the total number of patterns of that length. The "Count" column represents the total number of guesses made to find each of the 2^n patterns using the strategy above. If that number is divided by 2^n, we obtain the number in the "Average" column: the average number of guesses required to correctly identify (but not necessarily name) the secret

pattern. The "Maximum" column is the maximum number of guesses needed for that particular bit length, and is (perhaps surprisingly) the same as the number of bits in the pattern.

Obviously, if you announce that you're going to use this strategy, your opponent will pick one of the patterns that require the full n steps.

3 Equivalent Strategies

From the point of view of the game, there's no difference between a 1 and a 0 in a pattern: they might as well be two different colors, so the following strategy is exactly equivalent to the "Find the 1's" strategy, except it's a "Find the 0's":

The first guess is 00000000 and the result will be a number k that counts the number of 0's in the secret pattern. Then, successively guess 10000000, 01000000, 00100000, and so on, up to 00000010 which will identify the positions of the zeros in exactly the same way as before.

Are there any other strategies that are essentially equivalent to "Find the 1's"? The answer is a resounding *yes*! Here's an easy way make 40320 of them:

Once you know the count of the 1's you can search for them in any order, so, for example, this could be the sequence of guesses: 11111111, 11101111, 10101111, 10101110, $\ldots$, where we pick the seven bits we search for in an arbitrary order by adding 0's in that order through the pattern. The first one can be chosen in any of 8 positions; the second in any of 7 and so on, for a total of $8 \cdot 7 \cdot 6 \cdots 2 = 8! = 40320$ different strategies. (And you can trivially double this by doing the same thing with the "Find the 0's" strategy.

Can we do anything with the "Try a possible" strategy? Sure! Rather than just guess a possible string with the lowest number, choose one at random for a guess. It's a lot harder to count the number of strategies this way, since depending on the pattern and guess, the number of possible patterns does not go down in a predictable way. For example, if the first guess is 00000000 and the response is 4, there are $\binom{8}{4} = 70$ possible moves, but if there are only 2 zeros in the secret pattern, there are $\binom{8}{2} = 28$ choices for the next guess.

In any case, either of these two strategies provide lots of variation which makes it harder for the secret pattern chooser to avoid a rapid discovery of the secret.

4 Minimizing Strategy

The "Try a possible" strategy simply selects each successive guess to be the first available pattern (when considered to be a number) that has not been eliminated by the responses to the previous guesses. It seems like it might be interesting to choose a guess that is "the best" in some sense.

The following strategy, which we'll call the "Minimizing strategy" attempts to do this.

Think about the children's game called "Twenty questions" where you try to determine the type of animal that your opponent has secretly selected by asking yes/no questions. It seems obvious that you'd like to select questions that split the remaining possibilities into groups that are as equally-sized as possible. So it's much better to ask, "Is it a vertebrate?" rather than "Is it a lion?"

The same idea can be applied here, although the responses aren't just "yes" and "no", but a number between 0 and n. A perfect question would yield an equal number of responses in each of those $n + 1$ categories. (This is, of course, usually impossible, since there is at most one way to get a response of 0 and one way to get an n.) But we would like to split the

responses "as evenly as possible" into the $n+1$ piles.

We don't have the time to go into this in general, but the concept of "entropy" in information science is exactly what we want. In our case, if $p_0, p_1, \ldots, p_n$ are the probabilities of getting a response of getting $0, 1, \ldots, n$, respectively, then the entropy (usually called H) is given by the following equation:

$$H = -(p_0 \log p_0 + p_1 \log p_1 + \cdots + p_n \log p_n).$$

(Usually, the logarithm above is taken to be base-2, but that doesn't matter. Similarly, if any of the p_i are equal to 0, just assume that the term is 0 in spite of the fact that the logarithm of 0 is undefined.)

The distribution with the largest entropy is, in a very reasonable sense, the distribution that is most spread out.

So the Minimizing strategy works as follows:

1. Pick a first guess at random.
2. For every possible guess, divide the remaining possibilities into $n+1$ piles, where a possibility goes into pile k if that guess with that possibility yields a response of k. Calculate the entropy for that distribution.
3. Find the distribution above with the largest entropy and make that the next guess.
4. If the answer is determined, quit; otherwise, go back to step 2.

Obviously, there is vastly more calculation involved here, but here are the results of applying the Minimizing strategy to all possible target choices where the columns have exactly the same meanings as in Section sec:easystrategies.

n	Patterns	Count	Average	Maximum
1	2	2	1	1
2	4	6	1.5	2
3	8	18	2.25	3
4	16	44	2.75	4
5	32	108	3.375	4
6	64	254	3.96875	5
7	128	558	4.35938	6
8	256	1210	4.72656	6
9	512	2646	5.16797	7
10	1024	5638	5.50586	7
11	2048	12074	5.89551	7
12	4096	25526	6.23193	8
13	8192	53990	6.59058	8
14	16384	113180	6.90796	9

If you compare the two tables, it's easy to see that this one is better, at least if $n > 3$, and starting at $n = 5$ we see that the worst possible case is significantly improved upon. (You can also see that this table is significantly shorter than the earlier one since the computer time required to find all the entropies is vastly more for this situation.)

Also, as with the previous strategies, this one can be "encoded" by permuting the n bits, and/or by using the **xor** function to mask, say, an initial guess of 11111111.

One interesting thing about this strategy is that quite often the guess that maximizes the entropy is a pattern that is impossible, based on the previous answers. If we restrict the guesses to possible answers, the strategy works better than when we just choose an arbitrary (the smallest) possible answer, but not as well as this one, and it also appears that in the worst case, n guesses are required for n-bit games.

5 Computer Algorithm Considerations

How was the table in Section 2 calculated?

For small values of n almost any sort of coding will work, but if we wish to apply the strategy "Try a possible" to every one of the $2^{20} = 1048576$ possible secret patterns, a sloppy implementation can take a *long* time to run.

It clearly makes sense to store the patterns as a binary number so that only one bit per bit is required and so that we can work with an entire pattern as one "chunk". A 16- or 32-bit word is required, depending on how big n is. (Working with $n > 32$ makes most interesting problems intractable.)

5.1 Match Calculation

Suppose that the secret bit pattern is called S and the guess is called G. One operation that needs to be performed repeatedly is to find the number of bits of S that match the corresponding bit of G. There is a logical operation that is implemented on most computers in hardware called "Exclusive-Or", or **xor** for short. In the "C" computer language (and others) the **xor** operation is indicated by the circumflex symbol "^".

If we think of 1 as meaning "true" and 0 as meaning "false", then **xor** is an operation that takes two logical (true/false) values and returns true if exactly one of the two input values is true. Thus we have:

$$0 \hat{} 0 = 0;\ 0 \hat{} 1 = 1;\ 1 \hat{} 0 = 1 \text{ and } 1 \hat{} 1 = 0.$$

In computers, the **xor** operation works on every pair of corresponding bits in a computer word, so if it were applied to S and G, it would produce a word of the same size where there was a zero whenever the two patterns matched and a one where they didn't. For example, if $S = 10010110$ and $G = 00100011$ then $G \hat{} S = 10110101$. If we count the number of zeros, we'll know the number of matches! If we want to have a 1 where there's a match, just take the logical negation of the resulting word, meaning that we flip the value of every bit. The notation will be this: If $S = 01101011$, then we will write $\neg S = 10010100$.

Most computers do *not* have a hardware command to count the number of 0's or 1's in a computer word. The straightforward way to do this would be to successively check the final bit and shift right, adding one to the total if the right-most bit was 1. So for $n = 20$, there are up to 20 shifts and possible adds for each calculation. If you're going to do a lot of calculation, it's probably easier just to count all the bits in all the numbers from 0 to $2^n - 1$ once and store the results in an array called something like $\mathrm{bitcount}[]$.

In computerese, $\mathrm{bitcount}[!(\mathrm{G} \hat{} \mathrm{S})]$ will yield the number of matches between G and S. Since we're looking at relatively small values of n, the $\mathrm{bitcount}[]$ array can store each value in a byte (8 bits), which may save a lot of storage.

Even with these optimizations, there is a lot of work to apply the "Pick a possible" strategy to all possible secret words when n gets to be 20 or so. We have $2^{20} = 1048576$ so the strategy needs to be run against more than a million examples. To eliminate impossible patterns requires passes through a list that's initially more than a million long and although successive passes are faster with possibilities eliminated, there's still a lot of work. When you go from n to $n+1$ both of those numbers

double, plus there are more passes required, so you can expect the time for the $n+1$ calculation to take at least 4 times that required for n. With huge arrays, things may start swapping out of main memory, so things can get very ugly, depending on how much memory is available.

6 More Equivalent Strategies

Our first two strategies mirror typical human thought processes. It's easy to think of first counting the number of 1's (or equivalently, 0's) and then using various algorithms to find the particular 1 (or 0) bits. If you're playing against another human and he/she knows you think like this, it's very unlikely that they'll pick a pattern with a tiny number of 1's (or 0's). Assuming your first guess is nearly always 11111111, if your opponent picks all 0's or all 1's, you'll have the answer after a single guess.

What if they select a secret word with a single 1? You can Then effectively do a binary search for it by guessing 11110000 to see if it's in the first or second half. Then (supposing it's in the second half) trying 00001100 to get it down to two bits and one more guess will yield the answer. A similar strategy works fine for just one 0, of course.

Is there a good strategy if you know there are exactly two 1's (or 0's)?

If we consider an obvious human-like first move of 11111111, note that there are $\binom{8}{0} = \binom{8}{8} = 1$ ways to get zero or eight 1's. There are $\binom{8}{1} = \binom{8}{7} = 8$ ways to have one or seven 1's, and so on. This means that if your opponent knows you're going to count the 0's or 1's on the first pass, he should pick a pattern with 4 (or maybe 3 or 5) 1's to minimize the amount that you learn from your first guess.

But suppose you pick a starting guess at random from all $256 = 2^8$ possibilities. you'll still have $\binom{8}{0} = \binom{8}{8} = 1$ ways to get zero or eight matches. There are $\binom{8}{1} = \binom{8}{7} = 8$ ways to have one or seven matches, and so on. Now it's harder (for us humans) to think about general "matches" than concrete 0's or 1's, but there is the same amount of information; you just have to figure out how to use it, and your opponent will no longer be able to pick a pattern that's guaranteed to leave you with a large number of possibilities to search through.

What saves the day again is our old friend **xor**!

If we look at the mathematical properties of **xor**, there are a bunch of nice properties (these apply to bitwise whole-word operations as well as to single bits:

$$\begin{aligned} A \hat{} B &= B \hat{} A \\ (A \hat{} B) \hat{} C &= A \hat{} (B \hat{} C) \\ (A \hat{} B) \hat{} B &= A \\ \text{If A} \neq \text{B then (A} \hat{} \text{C)} &\neq (B \hat{} C) \end{aligned}$$

The first two are the usual commutative and associative properties, but the third and fourth are quite interesting. The third states that you can recover the original word A after an **xor** operation with B simply by re-applying the **xor** operation with B again. This can be interpreted in an even more interesting way: Suppose I have a word A that I would like to convert into a word B by applying the **xor** operation to A and some other word C. I basically want to solve the following equation for C:

$$A \hat{} C = B.$$

All I need to do is to **xor** A to both sides, and we'll have:

$$C = A \hat{} B.$$

This means that (for $n = 8$) I can choose a value of C such that $A \hat{} C$ will be any of the $256 = 2^8$ possible patterns. Each of the C values will be different, or the final property of **xor** above will be violated.

You may have to think about this for a bit, but the consequence is that you can use **xor** to encode the method you're really using. Let's go back to the "Find the 1's" strategy. Your plan is to guess, in order:

$$\begin{array}{c} 11111111 \\ 01111111 \\ 00111111 \\ 00011111 \\ 00001111 \\ 00000111 \\ 00000011 \\ 00000001 \end{array}$$

until you've identified all the 1's.

But what you can do is to choose a random pattern; say, $K = 11010110$, and **xor** that with all the guesses you'd use in your obvious strategy, yielding:

$$\begin{array}{lcll} 11111111 & \hat{} & 11010110 & = & 00101001 \\ 01111111 & \hat{} & 11010110 & = & 10101001 \\ 00111111 & \hat{} & 11010110 & = & 11101001 \\ 00011111 & \hat{} & 11010110 & = & 11001001 \\ 00001111 & \hat{} & 11010110 & = & 11011001 \\ 00000111 & \hat{} & 11010110 & = & 11010001 \\ 00000011 & \hat{} & 11010110 & = & 11010101 \\ 00000001 & \hat{} & 11010110 & = & 11010111 \end{array}$$

and your series of guesses will be chosen from the right-hand column, again, in order. The first response you get is the number of "matches" and then you move from left to right, searching for the particular matches. When you've got them all, you quit.

Let's look at an example. Suppose the secret pattern is 00111110. If we start making guesses from the right-most column above, we get these responses: $4, 3, 2, 1, 2, 1, 2, 3$ corresponding to: down, down, down, up, down, up, up, or a pattern of 11101000, as we saw in Section 2.

If G is a particular guess, K is our code, or random pattern (11010110 in this example), and S is the secret then the responses to our query for that G will be the number of 1's in:

$$(G \hat{} K) \hat{} \neg S = G \hat{} (K \hat{} \neg S).$$

So $(K \hat{} \neg S) = 11101000$ and if we **xor** that with K, we obtain: $\neg S = 11000001$ or $S = 00111110$, so we can recover the secret pattern using any strategy we know works, but encoded by an arbitrary pattern.

7 Fixed Guess Sequences

With the strategies we've examined up to now, some yield fairly quick results (the "Find a possible" strategy finds the secret message in approximately half the number of guesses as there are bits, at least for reasonably large n). But in the worst

case, the strategies all sometimes require n guesses for an n-bit secret pattern. Can we do better?

In fact, if $n < 5$, no matter what you do, you'll need n guesses in the worst case (proof?). But for $n = 5$, we can do a little better. Consider the following guess sequence: $g_1 = 00000$, $g_2 = 00011$, $g_3 = 00101$, $g_4 = 01001$. Let's check the results of using this sequence against all $2^5 = 32$ of the 5-bit secret patterns:

Pattern	g_1 $= 00000$	g_2 $= 00011$	g_3 $= 00101$	g_4 $= 01001$
00000	5*			
00001	4	4	4*	
00010	4	4	2*	
00011	3	5*		
00100	4	2	4*	
00101	3	3	5*	
00110	3	3	3	1*
00111	2	4	4*	
01000	4	2	2	4*
01001	3	3	3	5*
01010	3	3	1	3*
01011	2	4	2	4*
01100	3	1	3	3*
01101	2	2	4	4*
01110	2	2	2	2*
01111	1	3	3	3*
10000	4	2	2	2*
10001	3	3	3	3*
10010	3	3	1	1*
10011	2	4	2	2*
10100	3	1	3	1*
10101	2	2	4	2*
10110	2	2	2	0*
10111	1	3	3	1*
11000	3	1	1*	
11001	2	2	2	4*
11010	2	2	0*	
11011	1	3	1*	
11100	2	0*		
11101	1	1	3*	
11110	1	1	1*	
11111	0*			

The table shows the number of matches of every possible secret pattern with each of the four guesses. As soon as the sequence of numbers in a row is unique, an asterisk is added to the last number to indicate that it is no longer in doubt. After four guesses all are determined.

Obviously, you could take all four guesses and **xor** them with the same pattern to obtain another set of four guesses that would work equally well.

According to my computer simulations, 5 guesses are required if $n = 6$, 6 guesses for $n = 7$ and $n = 8$, and 7 guesses for $n = 9$ and $n = 10$.

If you have a solution using k guesses for n bits, it's easy to find a solution with $k + 1$ guesses for $n + 1$ bits. Because we have the ability to transform any good set with **xor**, assume that the first guess in the n-k solution is $000 \cdots 0$. To obtain the $(n + 1)$-$(k + 1)$ solution, append a zero to the beginning of each of the n-k guesses and add this one: $1000 \cdots 0$. For what it's worth, here are the solutions that it was hard to find:

For $n = 8$:

00000000
00000011
00000101
00011000
00101001
01001110

For $n = 10$:

0000000000
0000000111
0000011100
0000100101
0010101011
0101101110
0001110000

The question of what the minimum number of guesses are to assure a solution for the general n-bit game is open, as of the date this article was written.