Who Will Win the Toss?

H. Ramesh & V. Vinay

2010 AD. It is a bright sunny day at Lords. India and England are tied 1-1 and this is the deciding test. Geoff Boycott files in his pitch report: the side batting first has an enormous advantage as the wicket would probably crumple by the fourth day! Tendulkar and Hollioake walk out for the toss. The situation is tense. For purely jingoistic reasons, it is paramount for Tendulkar to win the toss. Of course, in 2010 there is no coin to toss! The ICC has just approved of a new product called `Source` which outputs a `h` or a `t` at random when a button is pressed. But, the Indian captain is not sure `Source` is fair; he shares the same skepticism that the Indian politician has for electronic voting. He confides this to Sunny, who is covering the toss. The technically correct Sunny (after prudently switching off the mike) shoots back, "Hmm. But what is a fair Source? How did the ICC certify the fairness of `Source`?"

Well, one way to find out is to press the `Source` button a number of times. If it comes up heads roughly half the times and tails the other half, it must be fair. If you think a little bit, then you will not be satisfied with this method. For, `Source` could give heads first, then tails, then heads, then tails, and so on. The sequence is always `hthththt....` Heads half the times and tails the other half. You can't call that fair, can you? So then, when can we call `Source` fair?

Intuitively, you know what property `Source` must have. If I invoke it a number of times, I should be unable to find any specific patterns in the sequence of heads and tails obtained. For example, `hththt` is a clear pattern, `httththttththttht` is a little less clear, but nevertheless a pattern. And a source which repeatedly produces this pattern cannot be fair.

So I press the `Source` button a large number of times. Now I look to see if there are any specific patterns in the sequence obtained. How do I check for the presence of a pattern? If there are no patterns, then `h` should occur half the time as should `t`. `hh, ht, th, tt` should each occur a fourth of the time, `hhh, hht, hth, htt, thh, tht, tth, ttt` should each occur an eighth of the time, and so on. In other words, each subsequence of length n should occur $1/2^n$th of the time. Another way to say this is that the probability that a particular subsequence of length n occurs is $1/2^n$, so all 2^n subsequences of length n are equally likely. What is more, this should hold for every n, not just for $n = 2$ or 3 or something. If this property indeed holds, then it looks like there are no patterns which one can find. Therefore, it seems as if I am justified in calling the sequence of heads and tails obtained by `Source`, random.

This seems to make sense. But just to be sure, let us examine a sequence which has the above property. Can you construct such a sequence? Constructing such a sequence is not obvious. A long time ago, a friend of Alan Turing at Cambridge by the name of David Champernowne invented (or discovered) precisely such a sequence. It goes by his name now. It is called Champ's number. Instead of heads and tails, the sequence comprises our familiar decimal digits

0-9. The sequence is simply 01234567891011121314151617...100101102... and so on. This indeed has the miraculous property that each subsequence of length n occurs $1/10^n$ (not 2^n now but 10^n) of the time. There are 10^n subsequences of length n and each is equally likely.

But now, we have cause for dissatisfaction again. How can one even call Champ's number a random number? Granted that each subsequence of length n appears the same number of times. But so what? The number has a clear pattern, so easy to predict. It is nowhere near random.

Then what makes a sequence random? Are numbers like $\sqrt{2}, \pi$, e all random. After all, they go on for ever without any noticeable pattern.

Mathematicians Steve Pinkus, Burton Singer and Rudolf Kalman have approximately determined the degree of randomness of $\sqrt{2}$, $\sqrt{3}$, π, and e, based on observing the frequencies of occurrence of various subsequences. They found that π was the most random of the lot, followed by $\sqrt{2}$, e, and $\sqrt{3}$ in that order. The frequency of occurrence of subsequences of length n is most equitable in π, a bit more disparate in $\sqrt{2}$, more so in e, and so on. And this is true for every value of n.

The Puzzle of the Random Sequence

When does one call a sequence random? This puzzled many for a while until some Mathematicians and Computer Scientists provided a really simple and elegant solution to this puzzle. The solution was the result of the work of Andrei Kolmogorov, Ray Solomonoff, Gregory Chaitin, Per Martin-Löf, and Leonid Levin, among others. It says:

Kolmogorov or Algorithmic Complexity

The Algorithmic Complexity of a sequence is the length of the **shortest program** which outputs that sequence. A sequence of a certain length is random if its Algorithmic Complexity is almost as big as the length of the sequence itself.

You might ask: what does a program have to do with a source being fair, a sequence being random? Why should the length of the program determine how random the sequence is? Why does the definition consider shortest programs and not just any program? And why does it consider the length of a program rather than its running time?

So let us examine this solution a bit. Given any sequence S, the following program seems like the shortest program to output S.

```
print(S);
```

But, no, there is a catch. This is not a small program! The entire sequence S has to be explicitly written out in the print statement. So if S is a sequence of length million, this program has length more than million. Thus the length of the program is a little more than the length of S itself, the little more being the characters taken by the `print` statement and the brackets. This is a pretty long program, longer than S itself. Can I write a shorter program? Looks like I cannot make it any shorter than the number of characters needed to specify the length of S. This is because this length has to figure somewhere in the program. For example, if I wanted a sequence of length 2000, I would need a program of length at least 4, probably longer accounting for the `print` statement and the punctuation. Let me take some examples to see whether I can get close to this figure.

Suppose one has the sequence `hthththt...` repeated a million times. There is a very small program which will output this sequence. While this sequence is a million long, the program length is just a few characters (49 to be precise), much smaller than the length of the sequence. Here is the program.

```
\{Program for hththt..\}
main(){
int i;
 for (i=0;i<500000;i++)
      printf(``th'');
}
```

If I want the program to output `ht` a trillion times, instead of a million, I need to change only the million in the program to a trillion. The other things remain the same. So to generate `ht` n times, a program of length which is roughly 50 more than the number of characters required to specify n suffices. This, in turn, is $\log_{10} n$ if numbers are specified in decimal. And $\log_{10} n + 50$ is much smaller than n when n is a large number like a 100 or 1000.

Of course, one could write much longer programs which output the same sequence. For instance, one could put any number of variable declarations above the *for* loop, with none of the variables being used in the program. The program would then be much longer, but it would output the same sequence. Therefore, to define how random a sequence is, one much necessarily take the length of the *shortest* program which outputs this sequence, and not just any program.

Next, suppose one wants to write a program to output a certain number of digits of Champ's number. Here is how the program would go. k in the program would depend upon how many digits of Champ's number are required. This program is even smaller than the program to output `hththt...`. So Champ's number has a very short program indeed and thus again does not qualify to be a random number.

```
\{Program For Champ's Number\}
for (i=0;i<k;++i)
   {printf(``\%d'',i)}
```

This definition really does seem to make sense, doesn't it? After all, Champ's number is really not random, it has an obvious pattern in it. And indeed it has a small program which generates it. How would one write programs to output irrational numbers like e? How long would these programs be? Remember the formula from school, $\mathrm{e} = 1 + \frac{1}{2!} + \frac{1}{3!} + \ldots$. This should be easy to program. One can program this with a `for` loop which repeats a number of times depending upon the number of digits of e required. Again this is a small program. I could output millions of digits of e by writing a program whose length is just a couple of hundred characters. So e is not very random as well, though a little more so than Champ's number.

Now this is puzzling, all sequences we know seem to have very small or at least not too large programs. It seems reasonable that a sequence with no patterns in it would require a larger program to generate than a sequence with patterns in it. But is there at all a random sequence on earth? A sequence with the following property: no matter what program I write to output this sequence, the length of the program should be comparable to the length of the sequence itself and not too much smaller.

Indeed, there are such sequences, but before I show you such a sequence, I will need to clarify some things.

First, what programming language and what computer are we talking about? In some programming languages, e.g., C, the length of a program I write may be shorter than the length of the same program in COBOL. So let us talk about programs in one language. For convenience, computer scientists like to work with a programming language in which each program is just a string of 0s and 1s. This might seem a bit strange but remember than the PC on your desk can understand only such programs. When you write programs in C, the compiler converts it to a program in machine code, which is just a stream of 0s and 1s.

Second, I will assume that no program in this programming language is a prefix of any other program in this programming language. So both 0110 and 011010 cannot be valid programs in this language. The first is a prefix of the second. This is essentially like saying that each program has a STOP command at the end. Since this command does not appear in the middle of any program, no program can be a prefix of another.

With this in mind, I can now show you that there is indeed a random sequence, one with large Algorithmic Complexity. However, I will not (not yet) show you such a string. Rather, I will show that such a string has to exist but without giving you an explicit example.

Let us consider all possible sequences of `h`s and `t`s, or of 0s and 1s, of a certain length, say n. There are 2^n such sequences. I will show you that at least one of them cannot be output by a program of length less than n. Why? Because there are only 2^i programs of length i (recall our programming language which we agreed upon a few minutes ago). The number of programs of length less than n is at most $2^0 + 2^1 + 2^2 + \cdots + 2^{n-1}$, counting programs of each length less than n. This sums to $2^n - 1$. Since each program can output only one string, all programs of length $n-1$ or smaller can only account for $2^n - 1$ of the 2^n sequences of length n. So there is at least one sequence of length n which requires a program of length n or more to generate. Voila!

What is more? Using the same argument you can show that half the sequences of length n can be generated only by programs of length $n-1$ or more. Therefore, there are a whole lot of strings which are substantially random.

At this point, you are probably a bit impatient. You know that there are random sequences. But you want a concrete example which will give a feel for what a random sequence looks like. It is not easy to generate an example of such a sequence. This is reasonable, after all such a sequence is a random sequence and therefore one should not be able to find patterns in it. And therefore, one may not be able to describe it very succinctly.

However, there are such sequences. To give you an example, I will have to digress first to give you the story of one of the greatest intellectual achievements of this century, the story of Gödel and Turing. We will ultimately come back and give you a clever example of such a sequence.

The Story of Gödel and Turing

In 1900, David Hilbert, a German mathematician, posed the following question. Is there a program, which will decide if a given statement is true or false?

To make sure that you get Hilbert's question right, let me give an example. Suppose I get a geometry question at school. For example, is the sum of the angles in a 300-sided regular polygon equal to 43640 degrees? It is not obvious whether this statement is true or false. Hilbert's question is whether one can write a program which will take such a statement as input, and decide whether it is true or false. Is there really such a program?

And why only geometry? I could have any set of axioms, which are statements which we all accept as true. And then I might want to know whether a particular statement follows from these axioms. For example, given the addition and multiplication tables of all integers, I might want to know whether there is a prime number between 200000 and 200040. This program will tell me whether such a prime number exists or not. For geometry, the axioms are the postulates of Euclid, one of which is that two straight lines are either parallel or intersect in exactly one point. (What are the others?)

Of course, the program has to be told what the axioms are. And then it should be able to tell whether a given statement is true or false, i.e., whether it or its negation[1] can be derived from the axioms. We need to assume, of course, that the axioms are *consistent.* So *either* the statement *or* its negation can be derived from the axioms, but certainly *not* both. Axioms which can derive both a statement and its negation make little sense. For example, the two axioms $1+1=2, 1+1=3, 2 \neq 3$ are inconsistent. The statements 'is 1+1=2?' and 'is $1+1 \neq 2$?' are both derivable from the axioms.

Hilbert's question was a challenging one. Until Gödel (1931) and Turing (1937) came along and showed that the answer to this question was really simple, *if you looked at it the right way.* But, if you think that they succeeded in writing such a program, you are wrong!

They did not succeed in writing Hilbert's program. Instead, they showed that **no such program could be written**. Interestingly, before they did do, many mathematicians (including Von Neumann) of that time were convinced that Hilbert's program did indeed exist.

Gödel showed that given the usual axioms of arithmetic (essentially the addition and multiplication tables of integers), he could construct a statement which said something very strange. It essentially said "If I am true, I am false, and if I am false, I am true". It did not say so directly, of course. This meaning was couched in terms of integers, addition and multiplication. If you just looked at the symbols in the statement, it would appear much like the statement about prime numbers I mentioned earlier. But if numbers could speak, this is what

[1]The negation of the statement "There is a prime number between 200000 and 200040" is the statement "There is no prime number between 200000 and 200040".

they would say. Now, since we have assumed that the axioms are consistent, it must be the case that neither this statement nor its negation can be derived from the axioms. Do you see why?

Here is the reason. If we could derive one of the two statements then we could derive the other as well. This is how Gödel has tailored the statement. But this means that the axioms are not consistent, which is a contradiction.

Now, what does this imply about Hilbert's question? Gödel's statement shows that Hilbert's program does not exist. Because if it did, then what would it say about Gödel's statement? Would it say it is true or would it say it is false? Neither answer is correct because neither the statement nor its negation can be derived from the axioms. The only recourse is to conclude that Hilbert's program does not exist.

Let me now set the stage for Turing's introduction by describing to you a program which does almost what Hilbert envisaged. If the statement given to it is such that either it or its negation can be derived from the axioms, then it will give the right answer. Otherwise, if the statement given to it is one like Gödel's statement, it will just go into an infinite loop, it will never stop. I must warn you at the outset that the running time of this program will be hopelessly slow, so if you want to use it to solve your geometry assignments, you might have to wait for a long long time. Nevertheless, it will give you the right answers sooner or later.

The program would be constructed as follows. How would you show that a statement is true? By giving a proof for it. A proof is just a series of inferences written out in some language, say English. Our program will just start generating all possible sequences of English sentences one by one. Not just proofs for this statement, but all sequences of English sentences. And not just grammatically correct ones. First, sequences with 1 character will be generated, then sequences with 2 characters, then 3, and so on. For each sequence of sentences generated, it will check whether these sentences constitute a correct proof for either the statement or its negation. When such a proof is found, the program will halt and give the appropriate answer. I can actually write this program in a few hours, it is probably just a few hundred lines long.

Enumerating sequences of English sentences one by one is just a mechanical process. Lot of them may be grammatically incorrect or just meaningless, but we will generate them all, one by one. That is easy enough to do, if we generate sequences in increasing order of length. Also, when a particular sequence of sentences has been generated, it is not hard to check whether it is correct and whether it proves either the statement at hand or its negation. So while generating a proof for the statement could be hard, but checking the correctness of a given proof is easy. Like writing exams is hard, checking exam papers is easy. But when will this process come to an end?

If the statement at hand or its negation is true, then there must be a proof to this effect of a certain length m. Since there are only a finite number of English proofs with length m or less, the right proof will be generated sooner or later. The program will check this proof, find all is well, stop, and give the appropriate answer. On the other hand, if you give Gödel's statement for which we know no proof exists, the program will just go on and on. It will never find a proof for either it or its negation.

But, as I said earlier, the program will be really slow. There could be 26^{100} sequences of English sentences of 100 letters, ignoring punctuation marks. This is a huge number. By the time the program gets to proofs which are even a page long, it will be many centuries. Nevertheless, it works, in principle.

Now, let me pose you a question. Given a statement, is there a way to decide whether the a) statement is true, b) its negation is true, or c) neither is true?

I can hear you saying: that is simple! Just run the above program. If it halts, then you have the right answer. If it doesn't, then you know that neither the statement nor its negation is true. This is almost correct. Except that here is a little catch. Suppose I run the program for a while and it hasn't halted as yet. Can I then conclude that option c) holds?

No, because there might be a proof for the statement or its negation which is really very long, and if allowed to run further, the program will discover this proof. So, there is no way for me to tell whether this program will halt ultimately or not. Therefore, I cannot say which of the three options is the right one.

But now you say: do the following. Why not write a second program P to decide whether the first program Q described above halts or not on the given statement. P takes the program Q and the given statement as input, and outputs the decision. This should be possible, after all, Q is only a few hundred lines long. If P concludes that Q does not halt, then option c) must hold. On the other hand, if it concludes that Q indeed halts, then one of options a) or b) holds. In this case, Q itself will tell which of a) or b) holds. Thus we'll know which of options a), b), c) is true.

That is a neat idea. This is where Turing enters the scene. Turing[2], one of the founders of the field of Computer Science, showed that the program you want to write cannot be written. He showed that there is no program P, which will take as input a program Q and an input x to Q, run for some time, and then stop and answer whether or not Q halts on input x. This problem is called the *Halting* problem. Turing showed using a very simple proof that there is no program which solves the Halting problem.

So there is no way to decide which of options a), b), c) is true. And there is no way to construct a program which will detect infinite loops in other programs. With this little digression, let us get back to the original issue.

An Example of a Random Sequence

I promised to give you an example of a fairly random sequence. My example will in fact be an infinite sequence of 0s and 1s. If you want to generate the first 1000000 bits of this sequence, you will need a program of length at least $1000000 - c$. And if you wanted the first 1000000000 bits of this sequence, you will need a program of length at least $1000000000 - c$. Here c is some fixed number. We will see how c comes about. It is like the 49 in the program for generating `hthththt....`

Recall our programming language that we agreed upon. Programs are strings of 0s and 1s, and no program is a prefix of another. Now some of these programs halt and some do not. Recall what we mean by a program outputting a sequence.

[2]Ironically, Computer Science existed even before modern computers did!

A program is said to to output a certain sequence if it runs for a while, outputs this sequence, and then halts. Therefore, each program which halts generates a sequence of a certain length. And programs which do not halt do not really output any sequence; they are just stuck in infinite loops.

Now, consider all possible programs. We will determine a number N by inspecting these programs. N is a sum of terms of the form $\frac{1}{2^l}$ for each program of length l that halts. Suppose there is 1 program of length 1, 2 of length 2, 3 of length 4, and 1 of length 5 which halt. Then this number will be $\frac{1}{2} + \frac{1}{2^2} + \frac{1}{2^2} + \frac{1}{2^4} + \frac{1}{2^4} + \frac{1}{2^4} + \frac{1}{2^5} + \cdots$, plus other terms for longer programs. There is a property called Kraft's Inequality which states that this number N is always between 0 and 1. This number N is the desired sequence.

But this is a number? Where is the sequence? Let me give you an example of how to represent a number as a sequence. Take $\frac{1}{2}$. This is just .1 in binary. Take $\frac{1}{2} + \frac{1}{2^2}$, this is just .11 in binary. And $\frac{1}{2} + \frac{3}{2^3} + \frac{1}{2^4} = .1111$ in binary. Similarly, the number I constructed above can be expressed in binary. Ignore the decimal point and it becomes a sequence of 0s and 1s.

If you are confused about converting N to a sequence, here is the trick. Look at each term $\frac{x}{2^y}$, where x is the number of programs of length y which halt. Express x in binary and put this binary number with its last digit at the yth location after the decimal. If $x = 2$ and $y = 4$, then x is 10 in binary and we need to place 10 with the 0 being at the 4th location after the decimal. And if $x = 3$ and $y = 10$, then we need to place 11 with the 1 on the right falling on the 10th location after the decimal.

But now you say: I have no clue why N should be quite random. Why should the shortest program to output the first 1000000 bits of N have length at least $1000000 - c$? In fact, there may be a simple short program P to compute the first n bits of N, for any value of n. P will just enumerate all programs of length n or less. For each such program Q, it will determine whether Q halts or not. Then it will construct N, by adding up $1/2^l$ for each program of length l that halts. Wouldn't this be a short program?

Let us see. Enumerating all programs of length up to n is easy. Remember that a program is just a binary string of 0s and 1s. All binary strings need not be programs, just as sequences of English characters do not always form valid C programs. For a binary string to be a valid program, it has to adhere to the syntax of our programming language. My program could enumerate all binary strings of length n, and for each check using a compiler whether its syntax is correct. This will give all programs of length n or shorter.

But how about checking whether a particular program halts. Remember Turing, and the Halting problem. Checking whether a program halts is impossible! No program P can check whether a given program Q halts. Therefore the program P cannot be written.

So it doesn't seem easy to write a short program to generate a certain number of bits of n. Then how does one show that N is random?

Showing that N is random is quite tricky. It is all based on a key property of N. N actually contains information about which programs halt and which do not.

Suppose you tell me what the first thousand bits of N are. Then I can easily figure out how many programs of length at most 1000 halt and how many do not. In fact, I can write a program which will extract this information from the bits of N. If you give me the first 2000 bits, then this program will be able to extract this information for programs of length up to 2000.

How you can extract this information will require some thought. You can easily see that all the information about how many programs of length n halt is present in the first n bits of N. This is because N was a sum of terms of the form $\frac{x}{2^y}$, where x is the number of programs of length y that halt. But does this mean that the number of programs of length n which halt can be deduced from the first n bits of N. This is like saying: I will give you the sum of three numbers and then ask you for what the three numbers were. This is not possible, in general. However, the property that no program is a prefix of another ensures that the numbers for the various lengths do not get all mixed up in N; each of these numbers can therefore be extracted.

So what I am saying is that the number N is just a clever and a succinct way of writing down which machines halt and which do not. This bypasses the need to solve the Halting problem. How does this property help in showing that N is quite random? Here is how.

Suppose there is a program P to generate the first n bits of N. Then you can actually write a program Q which will use P and the property that I just stated to output a sequence with Algorithmic Complexity more than n. The length of Q will be the length of P plus some value c, which is a fixed number independent of the value of n. Since the output of Q has Algorithmic Complexity more than n, the length of Q must exceed n. Therefore the length of P plus c must exceed n. In other words, the length of P is more than $n - c$.

I'll show you how to write Q so that it uses P to output a sequence with Algorithmic Complexity more than n. To write Q, you will use P as a subroutine, with the main routine having c characters. To describe the main routine I will have to hand-wave a bit here. You should be able to construct the precise picture from this general description if you think for a while.

The main routine first calls P to get the first n bits of N. It then figures out how many programs of length at most n halt and how many do not. From this information, it figures out which programs of length at most n halt and which do not. I will tell you how shortly. Let me proceed for the moment.

Then, for each such program which halts, the main routine figures out which sequence this program outputs. This can be done by just executing the program until it halts and noting down the output. How does the main routine execute a program, you might ask. If you study Computer Science, you will learn that one program can simulate another, just like a PC, which is really a complicated program, executes the simpler programs that you write and run on it. I must stress here that the fact that the program halts is important, otherwise, no matter how long this program is simulated, the main routine will never figure out what sequence this program generates.

Finally, the main routine determines a sequence which is different from all the sequences output by halting programs of length at most n. (If I give you a set of sequences and ask you to produce a sequence different from the ones I gave you, you could produce one easily, right?) So this is no problem. The main routine then outputs this sequence. This sequence must have Algorithmic Complexity more than n, because it is not generated by any of the programs of length at most n which halt.

So, in a nutshell. Once Q has the first n bits of N, it is able to figure out all sequences with Kolmogorov Complexity at most n. Then it generates some sequence different from all these and outputs it. The length of Q is at most the length of P plus the length of the main routine. The length of the main routine is some number c independent of n. Now take a look at the box above; we have shown that P must have length more than $n - c$.

Now I need to tell you how the main routine figures out which programs of length up to n halt and which do not. Remember that the main routine is now armed with the count C of the number of these programs which halt. It had extracted this information from the first n bits of N which P had provided. What it will do is generate each of these programs one by one as we did a little earlier. It will then simulate, i.e., execute, each of these programs one by one. Each program will be simulated until it halts. The main routine will keep a count of the programs that have been verified to halt. When this count reaches C, then the main routine knows that all programs of length at most n which halt have been discovered.

You might find something fishy here. Suppose the main routine picks up a program and starts executing it. And suppose this program never halts. Then the main routine will go into an infinite loop and never halt!

Indeed, this could happen. So we have to be clever in the way the programs are simulated. What the main routine will do is to run each program for 1 step. Then each program which has not yet halted is run for 2 steps. Then each program which has not yet halted is run for 3 steps, and so on. This way we make sure we do not get stuck in any one program waiting for it to halt, which it may never. This process continues until C programs have halted. Then the main routine knows that the remaining programs are those which never halt.

That is the whole proof. Gregory Chaitin, one of the founders of this definition of randomness, has actually written this main routine in LISP. It is just a few pages long and its size is independent of the value of n. You might want to write down the program Q to get the details clear and to figure out the value of c. But suppose c is, say, 200. Then the shortest program to output the first million bits of N has length at least a million minus 200. Therefore N is random, if not completely, at least to a very substantial extent.

Now, if you are wondering about the frequencies of occurrence of various subsequences in the sequence N, then note the following. Algorithmic Complexity is indeed a robust definition of randomness!

The number N actually has the property that all subsequences of any particular length appear with roughly equal frequency. This is true of any sequence whose Algorithmic Complexity is large.

The Toss

As Tendulkar walks out for the toss, he suspects that `Source` is fixed. But he knows what to do to cover himself. He tells Hollioake: let us press the button

1024 times and look at the sequence S generated. Let $K(S)$ be its Algorithmic Complexity. If $2^{1024-K(S)}$ is at least, say 4, then I win. Otherwise we press the button once more and whoever calls right wins. Now assuming that `Source` is indeed fixed, who do you think will win? Hollioake or Tendulkar?

If `Source` has been fixed, then the sequence generated by it will not be random. Therefore $K(S)$ would be much smaller than 1024. Even if it is 1022, then Tendulkar will win!

On the other hand, if $K(S)$ is more than 1022, then the sequence S is substantially random, and so `Source` looks to be more or less fair. Then Tendulkar and Hollioake each has roughly a half chance of calling right on the next press of the button.

Thus Tendulkar has covered himself against `Source`-fixing. There is one problem which he has forgotten though. How will he determine $K(S)$? He believes he can write a program to compute $K(S)$ from S. What do you think?

References

[1] G.J. Chaitin, Algorithmic Information Theory, Cambridge University Press, 1990.

[2] M. Li and P. Vitanyi, An Introduction to Kolmogorov Complexity and its Applications, Springer-Verlag, 1997.