The advantage of using “descriptions” of the form $(M, w)$ is that, since $w$ can be as long as necessary, we can often greatly simplify $M$ by putting any special cases, exceptions, conditions, or restrictions that would be difficult or inconvenient to store in $M$’s finite-state control into $w$ instead. For example, to generate Sipser’s string $A$, one might store its length in bits into $w$ (as a binary counter, 101000) and then design the Turing machine $M$ to generate alternating 0s and 1s, decrementing the counter after printing each new bit, and finally erasing the counter when it reaches 0.

In this case, the Turing machine is complicated enough that its encoding $(M)$ will be more than sixteen bits long, in which case $(M, w)$ will be more than forty bits long — longer than $A$ itself. But if we consider a string $A'$ of, say, two hundred thousand bits, alternating 0s and 1s, we can use exactly the same $M$ but give it $w' = 110000110101000000$ as its input. Now $(M, w')$ will be shorter than $A'$ if we can find a way of encoding $M$ into fewer than 99990 bits, which should be easy.

On the other hand, if $B'$ is a string of two hundred thousand bits that don’t follow any discoverable pattern, then its simplest description will feature a Turing machine that halts immediately on startup and an input string that is simply a copy of $B'$. This description will contain slightly more than two hundred thousand bits, so the descriptive complexity $K(B')$ of $B'$ will be greater than its length, while the descriptive complexity of $A'$ will be less than $|A'|$. So $A'$ is compressible and $B'$ is not.

Sipser is somewhat inconsistent in his use of the notation ‘$(M)$’ for the encoding of the Turing machine $M$. Sometimes he uses that notation to refer to the encoding before each bit has been doubled and the delimiter appended, sometimes after. For instance, he uses it the first way in the next-to-last sentence before Figure 6.22 but the other way inside the figure itself. Fortunately, the only theorem that is affected by this inconsistency is Theorem 6.26, and in the course of that proof Sipser carries out the necessary construction directly.

A skeptical reader might observe that determining the exact descriptive complexity of a string will be, shall we say, computationally intensive, to the point of being impossible in the real world for strings of any great length. That is an accurate observation and helps to explain the fact that all of the theorems we prove about descriptive complexity state upper bounds instead of enabling us to compute exact values.

However, the actual situation is even worse. $K$ is not a computable function at all, as Sipser asks the reader to prove in Problem 6.23 (page 271).

One might think that a Turing machine could compute $K$ for a particular input $x$ just by generating every possible description of the form $(M, w)$ of length less than $|x| + c$ (where $c$ is the length of the encoding of the trivial Turing machine that halts immediately on startup, as in the proof of Theorem 6.24), simulating $M$ on input $w$, and determining whether the tape contains $x$ when $M$ halts. If the possible descriptions are generated in ascending order by length, and lexicographically among descriptions of the same length, then the first description in which $M$, given input $w$, halts with $x$
on its tape must be the minimal description of \( x \), and its length, which can easily be determined, will be \( K(x) \).

The problem with this brute-force approach is that sometimes \( M \) doesn’t halt at all on input \( w \)! If the simulation never finishes, then the value of \( K(x) \) is never determined.

In the proof of Theorem 6.27, some readers may be concerned that a Turing machine that implements a Python interpreter would have to be very large and intricate indeed. We don’t care, though, so long as this Turing machine can correctly execute any Python program, even one more complicated than itself. For the proof of the theorem, it makes no difference if the constant \( c \) that represents the length of the encoding of the Python interpreter is a million, a septillion, a googol, or whatever. It’s a finite, fixed constant, and that’s all the theorem claims.

It may seem odd to say that randomly chosen long strings contain more information than patterned strings of the same length, because, in a slightly different sense, strings that are randomly chosen don’t carry any information at all, since they aren’t determined by the intentions and choice of an intelligent agent and so cannot convey any particular message. So how can they contain any information at all?

The “information” that is measured by descriptive complexity might better be called “unpredictability.” What high-information strings have in common with strings that are chosen randomly is that each successive bit is completely unpredictable. The “information” that each new bit brings as it arrives from the outside world is its own identity, which is something that one didn’t know before one saw the bit but does know afterwards. It contrasts with a highly predictable string such as 0101010101010101, in which one can anticipate that the next bit will be 0, and so the arrival of that bit tells one nothing new.

Some of the properties that hold for almost all strings are a little counterintuitive, and you may need to remind yourself that we are considering arbitrarily long strings before they start to make sense. For example, almost all strings contain as a substring a string of a million 0 bits in a row—the fraction of strings that manage to avoid containing such a substring approaches 0 as a limit as the length of those strings increases without limit.

In almost all strings, the fraction of bits that are 1 bits is greater than 49.99% but less than 50.01%. The fraction of strings that show a greater bias in either direction approaches 0 as a limit as the length of those strings increases without limit.