

LISP PROGRAM-SIZE COMPLEXITY¹

Applied Mathematics and Computation
49 (1992), pp. 79–93

G. J. Chaitin

Abstract

A theory of program-size complexity for something close to real LISP is sketched.

¹This paper should be called “On the length of programs for computing finite binary sequences in LISP,” since it closely follows references [3] to [6]. But that’s too long!

1. Introduction

The complexity measure used in algorithmic information theory [1] is somewhat abstract and elusive. Instead of measuring complexity via the size in bits of self-delimiting programs for a universal Turing machine as is done in [1], it would be nice to use instead the size in characters of programs in a real computer programming language.

In my book *Algorithmic Information Theory* [1] I make considerable use of a toy version of pure LISP constructed expressly for theoretical purposes. And in Section 5.1 [1], “Complexity via LISP Expressions,” I outline a theory of program-size complexity defined in terms of the size of expressions in this toy LISP. However, this toy LISP is rather different from real LISP; furthermore gruesome estimates of the number of S-expressions of a given size (Appendix B [1]) are required.

One can develop a theory of program-size complexity for something close to real LISP; we sketch such a theory here. It was pointed out in [2] that this is a straightforward application of the methods used to deal with bounded-transfer Turing machines in [3–6]. In fact, the results in this paper closely follow those in my two earliest publications on algorithmic information theory, the two AMS abstracts [3, 4] (also reprinted in [2]), but restated for LISP instead of bounded-transfer Turing machines.

2. Précis of LISP

Our basic LISP reference is [7]. So we have a LISP that includes integers. Otherwise, we pretty much restrict ourselves to the pure LISP subset of LISP 1.5, so there are no side-effects. In addition to the usual LISP primitives CAR, CDR, CONS, EQ, ATOM, COND, QUOTE, LAMBDA, NIL and T, we need some more powerful built-in functions, which are described below.

We shall principally be concerned with the size in characters of programs for computing finite binary sequences, i.e., bit strings. The convention we shall adopt for representing bit strings and character strings is as follows. A bit string shall be represented as a list of the integers 0 and 1. Thus the bit string 011 is represented by the 7-

character LISP S-expression ($0_{\sqcup}1_{\sqcup}1$). Similarly, a character string is represented as a list of integers in the range from 0 to 255, since we assume that each character is a single 8-bit byte. Notation: $|bit\ string|$ or $|character\ string|$ denotes the number of bits or characters in the string.

LENGTH

Given a list, this function returns the number of elements in the list (an integer).

FLATTEN

This is easy enough to define, but let's simplify matters by assuming that it is provided. This flattens an S-expression into the list of its successive atoms. Thus

```
(FLATTEN(QUOTE(A(BB(CCC)DD)E)))
```

evaluates to

```
(A BB CCC DD E)
```

EVAL

Provides a way to evaluate an expression that has been constructed, and moreover to do this in a clean environment, that is, with the initial association list.

EVLIS

Evaluates a list of expressions and returns the list of values. Applies `EVAL` to each element of a list and `CONS`'s up the results.

TIME-LIMITED-EVAL

This built-in function provides a way of trying to evaluate an expression for a limited amount of time t . In the limit as t goes to infinity, this

will give the correct value of the expression. But if the expression does not have a value, this provides a way of attempting to evaluate it without running the risk of going into an infinite loop. In other words, `TIME-LIMITED-EVAL` is a total function, whereas normal `EVAL` is a partial function (may be undefined and have no value for some values of its argument.)

(`TIME-LIMITED-EVAL` plays a crucial role in [1], where it is used to compute the halting probability Ω .)

CHARACTER-STRING

Writes a LISP S-expression into a list of characters, that is, a list of integers from 0 to 255. In other words, this built-in function converts a LISP S-expression into its print representation.

S-EXPRESSION

Reads a LISP S-expression from a list of characters, that is, a list of integers from 0 to 255. In other words, this built-in function converts the print representation of a LISP S-expression into the LISP S-expression. `CHARACTER-STRING` and `S-EXPRESSION` are inverse functions.

These two built-in functions are needed to get access to the individual characters in the print representation of an atom. (In [1] and [9, 10] we program out/define both of these functions and do not require them to be built-in; we can do this because in [1] atoms are only one character long.)

3. Definition of Complexity in LISP

$$H_{\text{LISP}}(x) \equiv \min_{\text{EVAL}(e)=x} |e|$$

That is, $H_{\text{LISP}}(x)$ is the size in characters $|e|$ of the smallest S-expression e (there may actually be several such smallest expressions) that evaluates to the S-expression x . We usually omit the subscript LISP.

Here e must only be self-contained LISP expressions without permanent side-effects. In other words, any auxiliary functions used must be defined locally, inside e . Auxiliary function names can be bound to their definitions (LAMBDA expression) locally by using LAMBDA binding. Here is an example of a self-contained LISP expression, one containing definitions of APPEND and FLATTEN:²

```
((LAMBDA (APPEND FLATTEN)
  (FLATTEN (QUOTE (A (BB (CCC) DD) E))))
 (QUOTE (LAMBDA (X Y) (COND ((ATOM X) Y)
  (T (CONS (CAR X) (APPEND (CDR X) Y))))))
 (QUOTE (LAMBDA (X) (COND ((ATOM X) (CONS X NIL))
  ((ATOM (CDR X)) X)
  (T (APPEND (FLATTEN (CAR X)) (FLATTEN (CDR X)))))))
)
```

The value of this expression is (A BB CCC DD E).

4. Subadditivity of Bit String Complexity

One of the most fundamental properties of the LISP program-size complexity of a bit string is that it is subadditive. That is, the complexity of the result of concatenating the non-null bit strings s_1, \dots, s_n is bounded by the sum of the individual complexities of these strings. More precisely,

$$H(s_1 \cdots s_n) \leq \sum_{k=1}^n H(s_k) + c, \quad (4.1)$$

where the constant c doesn't depend on how many or which strings s_k there are.

Why is this so? Simply, because we need only add c more characters in order to “stitch” together the minimal-size expressions for s_1, \dots, s_n into an expression for their concatenation. In fact, let e_1, \dots, e_n be the respective minimal-size expressions for s_1, \dots, s_n . Consider the following LISP expression:

$$(\text{FLATTEN}(\text{EVLIS}(\text{QUOTE}(e_1 \cdots e_n)))) \quad (4.2)$$

²Although we actually are taking FLATTEN to be built-in.

Recall that the built-in function `FLATTEN` flattens an S-expression into the list of its successive atoms, and that `EVLIS` converts a list of expressions into the list of their values. Thus this S-expression (4.2) evaluates to the bit string concatenation of s_1, \dots, s_n and shows that

$$H(s_1 \cdots s_n) \leq \sum_{k=1}^n |e_k| + 25 = \sum_{k=1}^n H(s_k) + c.$$

This S-expression (4.2) works because we require that each of the e_k in it must be a self-contained LISP S-expression with no side-effect; that is the definition of the LISP program-size complexity H_{LISP} . This S-expression also works because of the self-delimiting LISP syntax of balanced parentheses. That is, the e_k are separated by their delimiting parentheses in (4.2), and do not require blanks to be added between them as separators.

One small but annoying detail is that blanks **would** have to be added to separate the e_k in (4.2) if any two successive e_k were atoms, which would ruin our inequality (4.1). There is no problem if all of the e_k are lists, because then they all begin and end with parentheses and do not require added blanks. Does our LISP initially bind any atoms to bit strings, which are lists of 0's and 1's? Yes, because although the initial association list only binds `NIL` to `NIL` and `T` to `T`, `NIL` is the null bit string! That is why we stipulate that the bit strings s_k in (4.1) are non-null.

5. A Minimal Expression Tells Us Its Size As Well As Its Value

Symbolically,

$$H(x, H(x)) = H(x) + O(1).$$

That is, suppose that we are given a quoted minimal-size expression e for x . Then of course we can evaluate e using `EVAL` to get x . And we can also convert e into the list of characters that is e 's print representation and then count the number of characters in e 's print representation. This gives us $|e|$, which is equal to the complexity $H(x)$ of x .

More formally, let e be a minimal-size expression for x . Then the following expression, which is only c characters longer than e , evaluates to the pair consisting of x and $H(x)$:

```
((LAMBDA (X) (CONS (EVAL X)
                   (CONS (LENGTH (CHARACTER-STRING X))
                         NIL)))
 (QUOTE e))
```

Thus

$$H(x, H(x)) \leq H(x) + c.$$

Conversely, let e be a minimal-size expression for the pair consisting of x and $H(x)$. Then the following expression, which is only c' characters longer than e , gives us x :

```
(CAR e)
```

Thus

$$H(x) \leq H(x, H(x)) + c'.$$

The fact that an expression can tell us its size as well as its value will be used in Section 9 to show that most n -bit strings have close to the maximum possible complexity $\max_{|s|=n} H(s)$.

6. Lower Bound on Maximum Complexity

$$\max_{|s|=n} H(s) \geq n/8.$$

To produce each of the 2^n bit strings of length n , we need to evaluate the same number of S-expressions, some of which must therefore have at least $n/8$ characters. This follows immediately from the fact that we are assuming that each character in the LISP character set is a single 8-bit byte 0–255.

7. Upper Bounds on Maximum Complexity

Consider an arbitrary n -bit string s whose successive bits are b_1, \dots, b_n . Consider the LISP expression

$$(\text{QUOTE}(b_1 _ . . _ b_n))$$

Here the individual bits b_i in the bit string s of length n are separated by blanks. The value of this expression is of course the list of bits in s . Since this expression is $2n + 8$ characters long, this shows that $H(s) \leq 2n + 8$. Hence

$$\max_{|s|=n} H(s) \leq 2n + c.$$

Let us do slightly better, and change the coefficient from 2 to 1/2. This can be done by programming out hexadecimal notation. That is, we use the digits from 0 to 9 and the letters from A to F to denote the successive quadruples of bits from 0000 to 1111. Thus, for example,

$$\overbrace{((\text{LAMBDA}(X) \dots))}^{\text{big function definition}} (\text{QUOTE}(\text{F_F_F_F}))$$

will evaluate to the bit string consisting of sixteen 1's. Hence each group of 4 bits costs us 2 characters, and we have

$$\max_{|s|=n} H(s) \leq n/2 + c',$$

but with a much larger constant c' than before.

Final version: let us compress the hexadecimal notation and eliminate the alternating blanks. Form the name of an atom by appending the hexadecimal digits as a suffix to the prefix **HEX**. (The prefix is required because the name of an atom must begin with a letter, not a digit.) Then we can retrieve the hex digits packed into the name of this atom by using the **CHARACTER-STRING** built-in:

$$\overbrace{((\text{LAMBDA}(X) \dots))}^{\text{bigger function definition}} (\text{CHARACTER-STRING}(\text{QUOTE}_ \text{HEXFFFF}))$$

will again evaluate to the bit string consisting of sixteen 1's. Now each group of 4 bits costs us only one character, and we have

$$\max_{|s|=n} H(s) \leq n/4 + c'',$$

with a constant c'' that is even slightly larger than before.

8. Smoothness of Maximum Complexity

Consider an n -bit string S of maximum complexity. Divide S into $\lfloor n/k \rfloor$ successive nonoverlapping bits strings of length k with one string of length $< k$ left over.

Then, by the subadditivity of bit string complexity (4.1), we have

$$H(S) \leq \lfloor n/k \rfloor \max_{|s|=k} H(s) + \max_{|s|<k} H(s) + c,$$

where c is independent of n and k . Hence

$$\max_{|s|=n} H(s) \leq \lfloor n/k \rfloor \max_{|s|=k} H(s) + c'_k,$$

where the constant c' now depends on k but not on n . Dividing through by n and letting $n \rightarrow \infty$, we see that

$$\limsup_{n \rightarrow \infty} \frac{1}{n} \max_{|s|=n} H(s) \leq \frac{1}{k} \max_{|s|=k} H(s).$$

However, we already know that $\max_{|s|=n} H(s)/n$ is $\geq 1/8$ and $\leq 1/4 + o(1)$. Hence the limit of $\max_{|s|=n} H(s)/n$ exists and is $\geq 1/8$ and $\leq 1/4$. In summary, $\max_{|s|=n} H(s)$ is asymptotic from above to a real constant, which we shall henceforth denote by β , times n :

$$\begin{aligned} \max_{|s|=n} H(s) &\sim \beta n \\ \max_{|s|=n} H(s) &\geq \beta n \end{aligned} \tag{8.1}$$

$$.125 \leq \beta \leq .250$$

and whose value is an n -bit string, and an expression e' of size

$$\leq \max_{|s|=k} H(s)$$

whose value is an arbitrary k -bit string.

From the quoted expression e we immediately obtain its size

$$|e| = \text{LENGTH}(\text{CHARACTER-STRING}(e))$$

and its value $\text{EVAL}(e)$, as in Section 5. Note that $|e| - 18$ is exactly (9.1). Next we generate all character strings of size $\leq |e| - 18$, and use **S-EXPRESSION** to convert each of these character strings into the corresponding LISP S-expression. Then we use **TIME-LIMITED-EVAL** on each of these S-expressions for longer and longer times, until we find the given n -bit string $\text{EVAL}(e)$. Suppose that it is the j th n -bit string that we found to be the value of an S-expression of size $\leq |e| - 18$.

Finally we concatenate the j th bit string of size $n - k$ with the k -bit string $\text{EVAL}(e')$ produced by e' . The result is an n -bit string S , which by hypothesis by suitable choice of e and e' can be made to be any one of the 2^n possible n -bit strings, which turns out to be impossible, because it gives a LISP expression that is of size less than $\max_{|s|=n} H(s)$ for the n -bit string S .

Why?

The process that we described above can be programmed in LISP and then carried out by applying it to e and e' as follows:

$$\left(\overbrace{(\text{LAMBDA} (X Y) \dots)}^{\text{very big function definition}} \underbrace{(\text{QUOTE } e)}_{\substack{\text{eventually} \\ \text{gives arbitrary} \\ n - k \text{ bit string}}} \overbrace{e'}^{\substack{\text{directly} \\ \text{gives arbitrary} \\ k \text{ bit string}}} \right)$$

This LISP S-expression, which evaluates to an arbitrary n -bit string S , is a (large) constant number c' of characters larger than $|e| + |e'|$. Thus

$$H(S) \leq |e| + |e'| + c' \leq \left(\max_{|s|=n} H(s) - \max_{|s|=k} H(s) - c \right) + \left(\max_{|s|=k} H(s) \right) + c'.$$

And so

$$H(S) \leq \max_{|s|=n} H(s) - c + c' < \max_{|s|=n} H(s)$$

if we choose the constant c in the statement of the theorem to be $c' + 1$.

10. Maximum Complexity Strings Are Random

Consider a long n -bit string s in which the relative frequency of 0's differs from $1/2$ by more than ϵ . Then

$$H_{\text{LISP}}(s) \leq \beta H\left(\frac{1}{2} + \epsilon, \frac{1}{2} - \epsilon\right) n + o(n).$$

Here

$$H(p, q) \equiv -p \log_2 p - q \log_2 q,$$

where

$$p + q = 1, \quad p \geq 0, \quad q \geq 0.$$

More generally, let the n -bit string s be divided into $\lfloor n/k \rfloor$ successive k -bit strings with one string of $< k$ bits left over. Let the relative frequency of each of the 2^k possible blocks of k bits be denoted by p_1, \dots, p_{2^k} . Then let k and ϵ be fixed and let n go to infinity. If one particular block of k bits has a relative frequency that differs from 2^{-k} by more than ϵ , then we have

$$H_{\text{LISP}}(s) \leq \beta H\left(\frac{1}{2^k} + \epsilon, \frac{1}{2^k} - \frac{\epsilon}{2^k - 1}, \dots, \frac{1}{2^k} - \frac{\epsilon}{2^k - 1}\right) \frac{n}{k} + o(n).$$

Here

$$H(p_1, \dots, p_{2^k}) \equiv - \sum_{i=1}^{2^k} p_i \log_2 p_i,$$

where

$$\sum_{i=1}^{2^k} p_i = 1, \quad p_i \geq 0.$$

The notation may be a little confusing, because we are simultaneously using H for our LISP complexity measure and for the Boltzmann–Shannon entropy H ! Note that in the definition of the Boltzmann–Shannon H , any occurrences of $0 \log_2 0$ should be replaced by 0.

The Boltzmann–Shannon entropy function achieves its maximum value of \log_2 of the number of its arguments if and only if the probability distribution p_i is uniform and all probabilities p_i are equal. It follows that a maximum complexity n -bit string s must in the limit as n goes to infinity have exactly the same relative frequency of each possible successive block of k bits (k fixed).

How does one prove these assertions?

The basic idea is to use a counting argument to compress a bit string s with unequal relative frequencies into a much shorter bit string s' . Then the smoothness of the maximum complexity (8.1) shows that the original string s cannot have had maximum complexity.

For the details, see [5, 6]. Here is a sketch.

Most bit strings have about the same number of 0's and 1's, and also about the same number of each of the 2^k possible successive blocks of k bits. Long strings for which this is not true are extremely unusual (the law of large numbers!), and the more unequal the relative frequencies are, the more unusual it is. Since not many strings have this unusual property, one can compactly specify such a string by saying what is its unusual property, and which of these unusual strings it is. The latter is specified by giving a number, the string's position in the natural enumeration of all the strings with the given unusual property. So it boils down to asking how unusual the property is that the string has. That is, how many strings share its unusual property? To answer this, one needs estimates of probabilities obtained using standard probabilistic techniques; for example, those in [8].

11. Properties of Complexity That Are Corollaries of Randomness

Let's now resume the discussion of Section 8. Consider an n -bit string S of maximum complexity $\max_{|s|=n} H(s)$. Divide S into $\lfloor n/k \rfloor$ successive

k -bit strings with one $< k$ bit string left over. From the preceding discussion, we know that if k is fixed and we let n go to infinity, in the limit each of the 2^k possible successive substrings will occur with equal relative frequency 2^{-k} . Taking into account the subadditivity of bit string complexity (4.1) and the asymptotic lower bound on the maximum complexity (8.1), we see that

$$\beta n \leq H(S) \leq \binom{n}{k} \left(\sum_{|s|=k} H(s)/2^k \right) + o(n).$$

Multiplying through by k , dividing through by n , and letting n go to infinity, we see that

$$\beta k \leq \sum_{|s|=k} H(s)/2^k.$$

This piece of reasoning has a pleasant probabilistic flavor.

We can go a bit farther. The maximum $\max_{|s|=n} H(s)$ cannot be less than the average $\sum_{|s|=n} H(s)/2^n$, which in turn cannot be less than βn . Thus if we can find a single string of k bits with less than the maximum possible complexity, it will follow that the maximum is greater than the average, and thus also greater than βk . This is easy to do, for

$$H(\overbrace{000 \cdots 0}^{k \text{ 0's}}) \leq O(\log k) < \beta k$$

for large k . Thus we have shown that for all large k ,

$$\beta k < \max_{|s|=k} H(s).$$

In fact we can do slightly better if we reconsider that long maximum complexity n -bit string S . For any fixed k , we know that for n large there must be a subsequence 0^k of k consecutive 0's inside S . Let's call everything before that subsequence of k 0's, S' , and everything after, S'' . Then by subadditivity, we have

$$\begin{aligned} \beta n < H(S) &\leq H(S') + H(0^k) + H(S'') + c \\ &\leq H(S') + O(\log k) + H(S'') \end{aligned}$$

where

$$|S'| + |S''| = n - k.$$

It follows immediately that

$$\left(\max_{|s|=n} H(s) \right) - \beta n$$

must be unbounded.

12. Conclusion

We see that the methods used in references [3–6] to deal with bounded-transfer Turing machines apply neatly to LISP. It is a source of satisfaction to the author that some ideas in one of his earliest papers, ideas which seemed to apply only to a contrived version of a Turing machine, also apply to the elegant programming language LISP.

It would be interesting to continue this analysis and go beyond the methods of references [3–6]. The Appendix is an initial step in this direction.

Appendix

There is an intimate connection between the rate of growth of $\max_{|s|=n} H(s)$ and the number of $\leq n$ -character S-expressions.

Why is this?

First of all, if the number of $\leq n$ -character S-expressions is $< 2^k$, then clearly $\max_{|s|=k} H(s) > n$, because there are simply not enough S-expressions to go around.

On the other hand, we can use S-expressions as notations for bit strings, by identifying the j th S-expression with the j th bit string. Here we order all S-expressions and bit strings, first by size, and then within those of the same size, lexicographically.

Using this notation, by the time we get to the 2^{k+1} th S-expression, we will have covered all $\leq k$ -bit strings, because there are not that many of them. Thus $\max_{|s|=k} H(s) \leq n + c$ if the number of $\leq n$ -character S-expressions is $\geq 2^{k+1}$. Here c is the number of characters that must be added to the j th S-expression to obtain an expression whose value is the j th bit string.

So the key to further progress is to study how smoothly the number of S-expressions of size n varies as a function of n ; see Appendix B [1] for an example of such an analysis.

One thing that limits the growth of the number of S-expressions of size n , is “synonyms,” different strings of characters that denote the same S-expression. For example, $()$ and NIL denote the same S-expression, and there is no difference between $(A \sqcup B)$, $(A \sqcup \sqcup B)$ and $(A \sqcup \sqcup \sqcup B \sqcup)$. Surprisingly, the fact that parentheses have to balance **does not** significantly limit the multiplicative growth of possibilities, as is shown in Appendix B [1].

References

- 1 G. J. CHAITIN, *Algorithmic Information Theory*, Cambridge University Press, 1987.
- 2 G. J. CHAITIN, *Information, Randomness & Incompleteness—Papers on Algorithmic Information Theory*, Second ed., World Scientific, 1990.
- 3 G. J. CHAITIN, On the length of programs for computing finite binary sequences by bounded-transfer Turing machines, Abstract 66T-26, *AMS Notices* 13:133 (1966).
- 4 G. J. CHAITIN, On the length of programs for computing finite binary sequences by bounded-transfer Turing machines II, Abstract 631-6, *AMS Notices* 13:228-229 (1966).
- 5 G. J. CHAITIN, On the length of programs for computing finite binary sequences, *Journal of the ACM* 13:547-569 (1966).
- 6 G. J. CHAITIN, On the length of programs for computing finite binary sequences: statistical considerations, *Journal of the ACM* 16:145-159 (1969).
- 7 J. MCCARTHY et al., *LISP 1.5 Programmer's Manual*, MIT Press, 1962.

- 8 W. FELLER, *An Introduction to Probability Theory and Its Applications I*, Wiley, 1964.
- 9 G. J. CHAITIN, *LISP for Algorithmic Information Theory in C*, 1990.
- 10 G. J. CHAITIN, *LISP for Algorithmic Information Theory in SETL2*, 1991.