

CDM

Program Size Complexity

Klaus Sutner

Carnegie Mellon University

www.cs.cmu.edu/~sutner

Straight Line Programs

More Randomness

So far we have seen two approaches to randomness:

- Density: von Mises, all reasonable subsequences must have density $1/2$.
- Genericity: Martin-Löf, must survive a universal sequential test.

The Auswahlregel method of Mises does not quite work, and both his and Martin-Löf's approach do not really address the question of when a string of, say, 1000 bits is random.

There is an answer to this due to Kolmogorov and Chaitin that also uses computability but focuses on finite strings rather than infinite ones. The key idea is *program size complexity*: the size of the smallest program that generates the string.

First a harmless example.

Computing Polynomials

Consider an integer polynomial $f \in \mathbb{Z}[x]$.

Usually a polynomial is given as a coefficient list (a_d, \dots, a_0) as in

$$f(x) = a_d x^d + a_{d-1} x^{d-1} + \dots + a_1 x + a_0$$

Given $b \in \mathbb{Z}$ it is then easy to evaluate f at b . Customarily one uses Horner's rule to make sure the number of multiplications is no worse than the degree of f :

$$f(b) = ((\dots (a_d b + a_{d-1})b + a_{d-2})b + \dots)b + a_0$$

In general, Horner's rule is a good way to evaluate polynomials, However, for specific polynomials f there might be a better way to compute $f(b)$ than the general algorithm.

Straight Line Programs

Here is a very rudimentary type of program that can be used to evaluate polynomials. We will use the indeterminate x in the examples below, but think of x as a given integer.

Intuitively, a *straight-line program* is a program containing no conditionals and no loops; only arithmetic operations. Note that in this model we have to compute the coefficients from scratch.

Definition 1. A *straight-line program* U of length n is a sequence of n assignments of the form

$$v_1 = 1$$

$$v_2 = x$$

$$v_i = v_l \text{ op } v_r \quad \text{where } 0 \leq l, r < i \leq n.$$

The only allowed operations are $\text{op} = +, -, \times$. The output of the program is the value of v_n .

SLP and Polynomials

It is clear that any SLP computes a polynomial function: just substitute v_l or v_r for v_i everywhere, and the resulting expression is a polynomial in x .

Also, any polynomial can be computed by a SLP.

$v_1 = 1$	1
$v_2 = x$	x
$v_3 = v_1 + v_1$	2
$v_4 = v_2 * v_2$	x^2
$v_5 = v_4 * v_3$	$2x^2$
$v_6 = v_5 - v_1$	$2x^2 - 1$
$v_7 = v_6 * v_2$	$2x^3 - x$
$v_8 = v_7 + v_1$	$2x^3 - x + 1$

Short Programs

Sometimes a very short program can compute long polynomials (if they have a lot of structure that can be exploited).

$$1 + 8x + 28x^2 + 56x^3 + 70x^4 + 56x^5 + 28x^6 + 8x^7 + x^8$$

$v_1 = 1$	1
$v_2 = x$	x
$v_3 = v_2 + v_1$	$1 + x$
$v_4 = v_3 * v_3$	$(1 + x)^2$
$v_5 = v_4 * v_4$	$(1 + x)^4$
$v_6 = v_5 * v_5$	$(1 + x)^8$

SLP-Complexity

Definition 2. The **SLP-complexity** of a polynomial $f \in \mathbb{Z}[x]$ is the least n such that a straight-line program of size n computes f .

There is an important open problem about the number of roots of such polynomials (Smale's Fourth Problem).

Is the number of integer roots of an arbitrary integer polynomial f polynomially bounded by the SLP-complexity of f ?

I.e., is there some constant c such that

$$\# \text{ roots of } f \leq n^c$$

where n is the SLP-complexity of f , an arbitrary polynomial.

Roots of Polynomials

Roots of real polynomials have been studied to death, there is a rich literature out there.

One of my favorites:

N. Obreschkoff

Nullstellen Reeller Polynome

VEB Deutscher Verlag der Wissenschaften, 1963

But here we are dealing with integer roots and all bets are off.

Degrees

Note that the degree of a polynomial with SLP-complexity n can be much larger than n , so a simple degree argument won't work.

$$\begin{array}{ll} v_1 = 1 & 1 \\ v_2 = x & x \\ v_3 = v_2 * v_2 & x^2 \\ v_4 = v_3 * v_3 & x^4 \\ v_5 = v_4 * v_4 & x^8 \\ \vdots & \\ v_n = v_{n-1} * v_{n-1} & x^{2^{n-2}} \end{array}$$

Recall: Addition Chains

Remember 15-251?

Generating a term x^r is really the same as the old Addition Chain problem:

Remove x , subtraction and multiplication from our SLPs.

Then we can only compute integers, starting at 1 and using addition.

Somewhat surprisingly, it is quite difficult to compute the SLP-complexity of a plain integer in this reduced setting.

$$n = 30$$

$v_1 = 1$	1
$v_2 = v_1 + v_1$	2
$v_3 = v_2 + v_2$	4
$v_4 = v_3 + v_3$	8
$v_5 = v_4 + v_4$	16
$v_6 = v_5 + v_4$	24
$v_7 = v_6 + v_3$	28
$v_8 = v_7 + v_2$	30

This is the “obvious” algorithm; yields length 8.

A Better Program

But the complexity of $n = 30$ is actually lower:

$v_1 = 1$	1
$v_2 = v_1 + v_1$	2
$v_3 = v_2 + v_2$	4
$v_4 = v_3 + v_3$	8
$v_5 = v_4 + v_2$	10
$v_6 = v_5 + v_5$	20
$v_7 = v_6 + v_5$	30

SLPs versus Intuition

Developing good intuition for SLP programs is also quite hard, common sense often leads one astray.

For example, it might be tempting to think that without loss of generality step i looks like this:

$$v_i = v_{i-1} + v_r$$

So v_l can always be assumed to be the largest intermediate value so far.

Wrong!

But the least counterexample is quite large: the conjecture fails first for $n = 12509$.

Why SLPs?

It is natural and easy to generalize SLPs a bit:

- Input variables x_1, \dots, x_k (only on the right hand side).
- Internal variables v_1, \dots, v_n (both sides).
- Admissible operators op .

This is exactly the same as feedback-free circuits (where each gate has exactly two inputs).

Constructing small circuits is the same as constructing short SLPs.

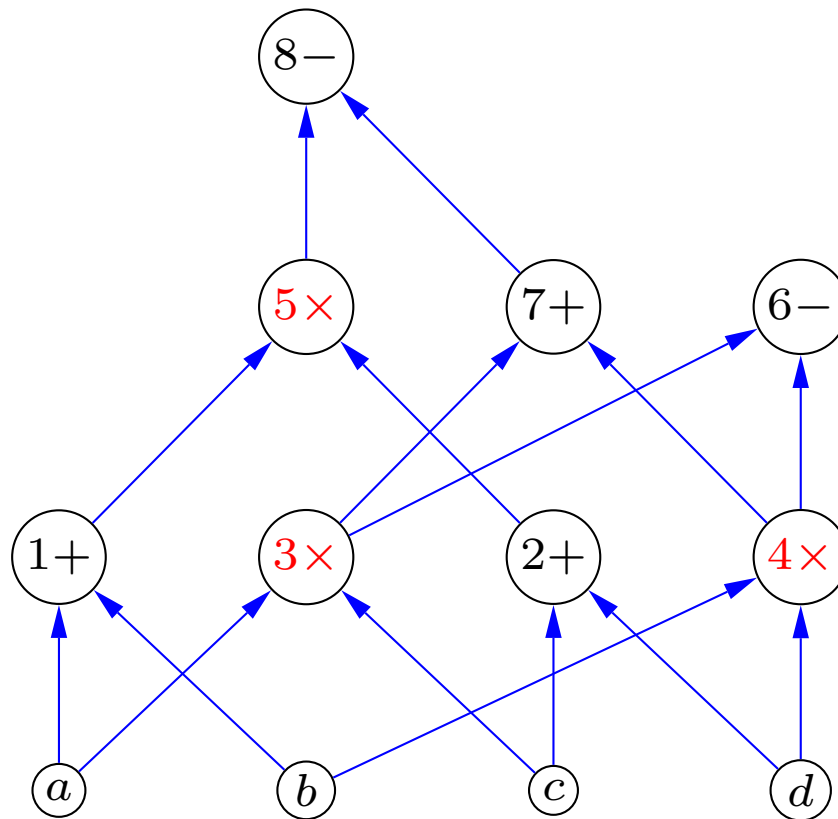
Classical Example: Complex Multiplication

$$(a + ib) \times (c + id) = (ac - bd) + (ad + bc)i$$

Seems to require 4 real multiplications. But it doesn't.

$v_1 = a + b$	$a + b$
$v_2 = c + d$	$c + d$
$v_3 = a * c$	ac
$v_4 = b * d$	bd
$v_5 = v_1 * v_2$	$ac + ad + bc + bd$
$v_6 = v_3 - v_4$	$ac - bd$
$v_7 = v_3 + v_4$	$ac + bd$
$v_8 = v_5 - v_7$	$ad + bc$

Multiplication Circuit



Extreme Compression

Here is another situation where the size of a program naturally appears as an interesting measure.

Suppose Ann wants to send Bob the first 10000 digits of π . In plain ASCII this requires transmission of 10000 bytes.

We could use some standard compression program to reduce the file size:

ASCII	10000 bytes
gzip	5106 bytes
bzip2	4369 bytes

Bob can then use the appropriate decompression program to retrieve the digits.

But there is a much better solution: don't send the digits, compressed or otherwise, but send a program that computes them.

A Pi Program

```
long a[35014], b, c = 35014, d, e, f = 1e4, g, h;

main()
{
    for( ; b=c-=14; h=printf("%04ld",e+d/f) )
        for( e=d%=f; g=--b*2; d/=g )
            d = d*b + f*( h ? a[b] : f/5 ), a[b] = d%--g;
}
```

After removal of all the superfluous white-space this is only 157 bytes long.

Of course, Bob has to work a bit harder (compile and execute).

Program-Size Complexity

Program-Size Complexity

Examples like the π program naturally lead to the question:

What is the shortest program that generates some specific output?

To obtain a clear quantitative answer, we need to fix a programming language and everything else that pertains to compilation and execution.

Then we can speak of the *shortest program* (in length-lex order) that generates some fixed output.

Note: This is very different from resource based complexity measures (running time or memory requirement). We are not concerned with the time it takes to execute the program, nor with the memory it might consume during execution.

Kolmogorov-Chaitin Complexity

The most natural setting for program size complexity is actually based on machines rather than programming languages (Turing machines), but for our purposes it suffices to think of

- C programs
- being compiled on a standard compiler,
- and executed in some standard environment.

So we have some program p and a compilation/execution system U (for universal machine).

We feed p to U and obtain the desired output.

Convention:

We may safely assume that the program and the output are both binary strings, so we are only dealing with words over $\mathbf{2} = \{0, 1\}$.

The Definition

Definition 3. For any word $x \in \mathbf{2}^*$, denote \hat{x} the length-lex minimal program that produces x on U : $U(\hat{x}) = x$.

The **Kolmogorov-Chaitin complexity** of x is defined to be the length of the shortest program which generates x :

$$C(x) = |\hat{x}| = \min(|p| \mid U(p) = x)$$

This concept was discovered independently by Kolmogorov, Solomonov and Chaitin.

Note that running an arbitrary program p on U may produce no output: the program may simply fail to halt. As we will see shortly, this is a huge problem.

Example 1. Let x be the first 100000 binary digits of π . Then x has Kolmogorov-Chaitin complexity at most a few hundred.

The Basics

Recall a comment some time ago that we can always program out ranking/unranking functions by hard-wiring a table into the program. The same is true here.

To see that \hat{x} and therefore $C(x)$ exists for all x store x as part of the program.

```
bit xarr[] = { x1, x2, ..., xn };  
print xarr[];
```

`xarr []` is an array of bits, corresponding to string x . Hence

$$C(x) \leq |x| + c$$

Cheating

Also note: we can cheat and hardwire any specific string x of very high complexity in U into a modified environment U' .

Let's say

- U' on input 0 outputs x .
- U' on input $1p$ runs program $U(p)$.
- U' on input $0p$ returns no output.

Then U' is perfectly reasonable, except that x has a fraudulently low complexity.

Clearly there is cheating going on, but it is much less clear how one could rule out all attempts at cheating. What is a “natural” universal machine?

Finite Cheating

The same argument applies to a finite collection of strings x_1, \dots, x_n : we can make sure that their complexity in U' is bounded by a constant.

But beyond this constant fudge factor, the choice of U doesn't matter much. If we pick another environment U' and define C' accordingly, we have

$$C'(x) \leq C(x) + c$$

since U can simulate U' using some program of constant size.

So we have robustness – which is a good thing since otherwise the definition would be more or less useless.

Concrete U



Greg Chaitin has actually implemented such environments U . He uses LISP rather than C, but that's just a technical detail (actually, he has written LISP interpreters in C). So in some simple cases one can actually determine precisely how many bits are needed for \hat{x} .

Properties

Proposition 1. *For any positive integer x : $C(x) \leq \log x + c$.*

This is just plain binary expansion: we can write x in

$$n = \lfloor \log_2 x \rfloor + 1$$

bits.

But note that for some x the complexity $C(x)$ may be much smaller than $\log x$.

For example $x = 2^{2^k}$ or $x = 2^{2^{2^k}}$ requires far fewer bits.

Exercise 1. *Construct some other numbers with small Kolmogorov-Chaitin complexity.*

Computable Functions

How about duplicating a string? What is $C(xx)$?

It is clear that we can construct a constant size program that will take as input a program for x and produce xx instead. Hence $C(xx) \leq C(x) + O(1)$.

Similarly we could generate x^r instead of x with constant overhead.

But here is a more surprising fact: we can apply any computable function to x , and increase its complexity by only a constant.

Lemma 1. *Let $f : 2^* \rightarrow 2^*$ be computable.*

Then $C(f(x)) \leq C(x) + c$.

Say What?

The last lemma is a bit hard to swallow, but it's quite correct.

Take your favorite exceedingly-hard-to-compute recursive function, say, the Ackermann function $A(x, y)$.

As we have seen, $A(10, 10)$ is a mind-boggling atrocity; much, much larger than anything we can begin to make sense of.

But

$$C(A(10, 10)) \leq \log 10 + \text{a little}$$

Some Exercises

Exercise 2. *Prove the two lemmata.*

Exercise 3. *Express the complexity of a concatenation xy in terms of the complexity of x and y .*

Exercise 4. *Is it possible to cheat in infinitely many cases? Justify your answer.*

Exercise 5. *Use Kolmogorov-Chaitin complexity to show that the language $L = \{ xx^r \mid x \in 2^* \}$ of even length palindromes cannot be accepted by a finite state machine.*

Compression

$C(x)/|x|$ is the ultimate compression ratio: there is no way we can express x as anything shorter than $C(x)$ (at least in general; recall the comment about cheating).

An algorithm that takes as input x and returns as output \hat{x} is the dream of anyone trying to improve gzip or bzip2.

Well, almost. In a real compression algorithm the time to compute \hat{x} and to get back from there to x is also very important. In our setting time complexity is being ignored completely.

As we will see, there is also the slight problem that $C(x)$ is not computable, much less \hat{x} .

Incompressibility

As is the case with compression algorithms, even C cannot always succeed in producing a shorter string.

Definition 4. A string $x \in 2^*$ is **c -incompressible** if $C(x) \geq |x| - c$ where $c \geq 0$.

Hence if x is c -incompressible we can only shave off at most c bits when trying to write x in a more compact form: an incompressible string is generic, it has no special properties that one could exploit for compression.

The upside is that we can adopt incompressibility as a definition of randomness for a finite string – though it takes a bit of work to verify that this definition really conforms with our intuition. For example, such a string cannot be too biased.

Having incompressible strings can be very useful in lower bound arguments: there is no way an algorithm could come up with a clever data structure that represents these strings.

Existence

How do we know that incompressible strings exist? By counting: there aren't enough short programs to generate all long strings. Here is a striking result whose proof is also a simple counting argument.

Lemma 2. *Let $S \subseteq 2^*$ be a non-empty set of words of cardinality n . For all $c \geq 0$ there are at least $n(1 - 2^{-c}) + 1$ many words x in S such that*

$$C(x) \geq \log n - c.$$

Example 2. *Consider $S = 2^k$ so that $n = 2^k$. Then by the lemma most words of length k have complexity at least $k - c$, so they are c -incompressible.*

Example 3. *Pick size s and let $S = \{0^i \mid 0 \leq i < s\}$. Specifying $x \in S$ comes down to specifying the length $|x|$. Writing a program to output the length will often require close to $\log s$ bits.*

But is it True?

This lemma sounds utterly wrong: why not simply put only simple words (of low Kolmogorov-Chaitin complexity) into S ? There is no restriction on the elements of S .

Since we are dealing with strings there is a natural, easily computable order: length-lex. Hence there is an enumeration of S :

$$S = w_1, w_2, \dots, w_{n-1}, w_n$$

Given the enumeration we need only some $\log n$ bits to specify a particular element. The lemma says that for most elements of S we cannot get away with much less.

Exercise 6. *Try to come up with a few “counterexamples” to the lemma and understand why they fail.*

The Proof

Proof.

Proof is by very straightforward counting. Let's ignore floors and ceilings.

The number of programs of length less than $\log n - c$ is bounded by

$$2^{\log n - c} - 1 = n2^{-c} - 1.$$

Hence at least

$$n - (n2^{-c} - 1) = n(1 - 2^{-c}) + 1$$

strings in S have complexity at least $\log n - c$.

□

Observation

It gets worse: the argument would not change even if we gave the program p access to a database D (some arbitrary string).

This observation is totally amazing: we could concatenate all the words in S into a single string

$$D = w_1 \dots w_s$$

that is accessible to p .

However, to extract a single string w_i we still need some $\log s$ bits to describe the first and last position of w_i in D .

Unbounded Complexity

As one would expect, sufficiently long strings have large complexity:

Lemma 3.

The function $x \mapsto C(x)$ is unbounded.

Actually, even $x \mapsto \min(C(z) \mid x \leq_{\parallel} z)$ is unbounded.

Here $x \leq_{\parallel} z$ refers to length-lex order.

So even a trivial string $000 \dots 000$ has high complexity if it's just long enough.

Example 4. *Consider strings x of 0 of length $2 \uparrow\uparrow k$ (k -fold iterated exponential). Then $C(x) \leq C(k) + c$ which is much less than $|x|$ but still tends to infinity.*

Halting

As mentioned, we may have that $U(p)$ is undefined simply because the program p never halts. And, since the Halting Problem is undecidable, there is no systematic way of checking:

Problem: **Halting Problem for U**

Instance: Some program $p \in 2^*$.

Question: Does p (when executed on U) halt?

Of course, this version of Halting is still semi-decidable, but that's all we can hope for.

Kolmogorov-Chaitin Complexity is not Computable

Theorem 1. *The function $x \mapsto C(x)$ is not computable.*

Proof. Suppose C is computable by some procedure $\text{comp}_{\text{kc}}()$. Consider the following algorithm A with input n (the loop is supposed to be in length-lex order).

```
foreach  $x \in 2^*$  do  
    let  $m = \text{comp}_{\text{kc}}(x)$ ;  
    if  $n \leq m$  then return  $x$ ;
```

A halts on all inputs n , and returns the length-lex minimal word x of Kolmogorov complexity at least n . But then

$$n \leq C(x) \leq C(n) + c \leq \log n + c',$$

contradiction. □

The Crux of the Matter

Let's try to pin down the problem with computing Kolmogorov-Chaitin complexity.

- Given a string x of length n we would look at all programs p_1, \dots, p_N of length at most $n + c$.
- We run all these programs on U , in parallel.
- At least one of them, say, p_i , must halt on output x .
- Hence $C(x) \leq |p_i|$.

But unfortunately, this is just an upper bound: later on a shorter program p_j might also output x , leading to a better bound.

But other programs will still be running; as long as at least one program is still running we only have a computable approximation, but we don't know whether it is the actual value.

Halting, Again

We could compute Kolmogorov complexity if someone gave us a black box for the Halting problem.

- Given a string x of length n determine all programs p of length at most $n + c$ such that $U(p)$ halts (use black box).
- Run all these programs on U , in parallel, until they all halt.
- Select that ones that output x .
- Pick the length-lex shortest one.

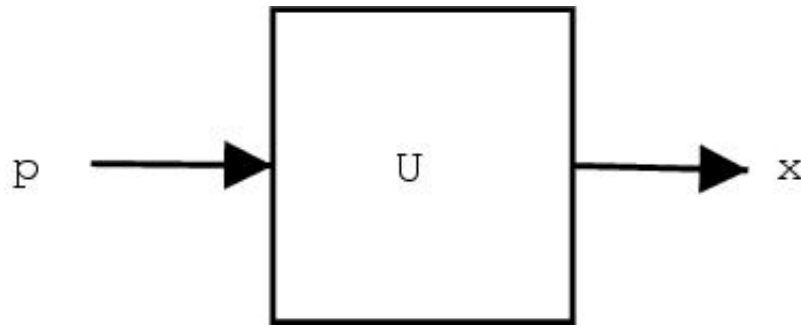
Of course, the first step is not algorithmic.

Note: This is exactly the same problem as with the Busy Beaver function.

Prefix Complexity

Prefix Programs

It turns out to be a bad idea to allow the universal machine U to operate on arbitrary input strings $p \in \mathbf{2}^*$.



Better results can be obtained if we insist that the collection of all syntactically correct programs is a prefix set: no program is a prefix of another.

Note that this condition trivially holds for most ordinary programming languages (at least in spirit).

Prefix Complexity

Definition 5. Let U_{pre} be a universal Turing machine which converges only on a prefix set $P \subseteq 2^*$ of strings (the admissible programs for U). Define the prefix Kolmogorov-Chaitin complexity of a string x by

$$K(x) = \min(|p| \mid U_{\text{pre}}(p) = x)$$

Note that in general $K(x) > C(x)$: there are fewer programs available so in general the shortest program for a fixed string will be longer than in the unconstrained case. Other than that, though, many results carry over. In particular

Corollary 1. $K(x)$ is not computable.

Exercise 7. Prove the last result (check that the proof for plain KC complexity carries over).

Print x Revisited

Recall that for ordinary Kolmogorov-Chaitin complexity it is easy to get an upper bound for $C(x)$: the program

```
print  $x_1x_2 \dots x_n$ 
```

does the job.

But if we have to deal with the prefix condition things become a bit more complicated. We could use delimiters around x , but remember that our alphabet is just $\mathbf{2} = \{0, 1\}$.

In the absence of delimiters, we need to use some construct such as

```
print next  $n$  bits  $x_1x_2 \dots x_n$ 
```

Of course, everything is coded up as bits, so we need to take a closer look at how this coding would work to get a bound on $K(x)$.

A Standard Prefix Code

Here is a simple way to satisfy the prefix condition: code a bit string as

$$E(x_1 \dots x_n) = x_1 0 x_2 0 \dots x_{s-1} 0 x_n 1$$

so that $|E(x)| = 2|x|$.

Of course, there are other obvious solutions such as $0^{|x|}1x$.

Both approaches double the length of the string, which doubling would lead to a rather crude upper bound $2n + O(1)$ for the prefix complexity of a string via the program

```
print E(x)
```

Can we do better?

Improving the Prefix Code

How about leaving $x = x_1x_2 \dots x_n$ unchanged, but using E to code $n = |x|$, the length of x :

$$E(|x|) x$$

Note that this still is a prefix code and we now only use some $2 \log n + n$ bits to code x .

But why stop here? We can also use

$$E(|x|) |x| x$$

This requires only some $2 \log \log n + \log n + n$ bits.

Iteration

In fact, we can iterate this coding operation. Let

$$E_0 = E$$

$$E_{i+1}(x) = E_i(|x|) x$$

Exercise 8. *Show that all the E_i are prefix codes. What is the optimal choice of k so that $E_k(x)$ has minimal length?*

But note that the right choice of k depends on the length of x . What we really need is a “infinity code” E_∞ that works for all x .

Taking Things to the Limit

Here it is:

$$E_{\infty}(x) = \text{len}_k(x) 0 \text{len}_{k-1}(x) 0 \dots |x| 0 x 1$$

where $k = \text{len}^*(x)$ and

$$\text{len}_1(x) = |x|$$

$$\text{len}_{i+1}(x) = |\text{len}_i(x)|$$

$$\text{len}^*(x) = \min(i \mid \text{len}_i(x) = 2)$$

Example 5. For a bit-string x of length 20000 we obtain the following $\text{len}_i(x)$:

$$100111000100000, 1111, 100, 11, 10$$

So the length of $E_{\infty}(x)$ is $20000 + 32$.

The Infinity Code

How much do we have to pay for a prefix version of x ? Essentially a sum of iterated logs.

Lemma 4.

$$|E_\infty(x)| = n + \log n + \log \log n + \log \log \log n \dots + \log^*(n) + O(1)$$

So this is an upper bound on $K(x)$.

Of course, some other coding scheme might produce even better results.

A good rough approximation to $K(x)$ is $n + \log n$, in perfect keeping with our intuition about

print next n bits $x_1x_2 \dots x_n$

Why Bother?

It's clear that prefix complexity is a bit harder to deal with than ordinary Kolmogorov-Chaitin complexity. What are the payoffs?

For one thing, it is much easier to combine programs.

For example, if p and q are prefix programs then the concatenation pq is uniquely parsable.

Hence we can use a program of the form Hpq where H is some special header to compute a composite string xy where p produces x and q produces y .

This operation is actually more complicated in the unconstrained case.

But the real reason for the prefix version is the following.

Chaitin's Ω

Definition 6. The total halting probability of any prefix program is defined to be

$$\Omega = \sum_{U_{\text{pre}}(p) \downarrow} 2^{-|p|}$$

Ignoring the motivation behind this for a moment, note that this definition works because of the following bound.

Lemma 5. *Kraft Inequality*

Let $S \subseteq \mathbf{2}^*$ be a prefix set. Then $\sum_{x \in S} 2^{-|x|} \leq 1$.

Exercise 9. Prove the Kraft inequality. As a warm-up, consider $S = \{0, 1\}^n$ or $S = \{0^i 1 \mid i \geq 0\}$.

But Why?

We can define the halting probability for a single target string x to be

$$P(x) = \sum_{U_{\text{pre}(p)}=x} 2^{-|p|}.$$

and extend this to sets of strings by adding: $P(S) = \sum_{x \in S} P(x)$.

Then $\Omega = P(\mathbf{2}^*)$.

Proposition 2. Ω is a real number and $0 < \Omega < 1$.

In fact, in a standard environment one can show (with quite some pain) that

$$0.00106502 < \Omega < 0.217643$$

Randomness

Proposition 3. Ω is incompressible in the sense that $K(\Omega[n]) \geq n - c$, for all n .

As a consequence, (the binary expansion of) Ω is Martin-Löf random.

This may seem a bit odd since we have a perfectly good definition of Ω in terms of a converging infinite series. But note the Halting Problem lurking in the summation – from a strictly constructivist point of view Ω is in fact not properly defined at all.

Halting and Ω

Lemma 6. Consider $q \in 2^n$. Given $\Omega[n]$ it is decidable whether U_{pre} halts on input q .

Proof.

Start with approximation $\Omega' = 0$.

Dovetail computations of U_{pre} on all inputs.

Whenever convergence occurs on input p , update the approximation: $\Omega' = \Omega' + 2^{-|p|}$.

Stop as soon as $\Omega' \geq \Omega[n]$. Then

$$\Omega[n] \leq \Omega' < \Omega < \Omega[n] + 2^{-n}.$$

But then no program of length n can converge at any later stage. □

If Only . . .

For $n \approx 10000$, knowledge of $\Omega[n]$ would settle, at least in principle, several major open problems in Mathematics such as the Goldbach Conjecture or the Riemann Hypothesis:

These conjectures can be refuted by an unbounded search, and the corresponding Turing machine can be coded in 10000 bits.

For example, here is the Goldbach conjecture:

Conjecture: Every even number larger than 2 can be written as the sum of two primes.

We can easily construct a small Turing machine that will search for a counterexample to this conjecture, and will halt if, and only if, the Goldbach conjecture is false.

Time Complexity

Of course, we don't have the first 10000 bits of Ω , nor will we ever.

Even if we did, it would take a long time to exploit this information: the running time of the oracle algorithm above is not bounded by any recursive function of n .

Another pretty application of Ω in number theory:

There is an exponential Diophantine equation

$$E(n, x_1, x_2, \dots, x_m) = 0$$

such that there are infinitely many solutions x_1, \dots, x_m iff the n -th bit of Ω is 1.

Loosely speaking: randomness lurks even within integer polynomials.

Summary

- Program size complexity tends to be very complicated, even when restricted to very simple programs such as SLPs.
- Kolmogorov-Chaitin complexity describes the ultimate, theoretic limit on compression.
- Some of the results in Kolmogorov-Chaitin complexity are a bit counterintuitive.
- Kolmogorov-Chaitin complexity can be used to give a precise meaning to randomness or lack-of-order. Has close connections to entropy.
- Kolmogorov-Chaitin complexity has become a very useful tool in complexity theory. E.g., lower bound arguments often use incompressible strings.
- Chaitin's Ω has become very popular as an example of an “unknowable” object in mathematics.