Program Size Complexity

Klaus Sutner
Carnegie Mellon University
Spring 2022

Where Are We? 2

- Kolmogorov-Chaitin algorithmic information theory provides a measure for the "complexity" of a bit string (or any other finite object). This is in contrast to language based models that only differentiate between infinite collections.
- Since the definition is closely connected to Halting, the complexity function $K(x)$ fails to be computable, but it provides an elegant theoretical tool and can be used in lower bound arguments.
- And it absolutely critical in the context of randomness; more later.

A Nuisance 3

Recall that our model of computation used in Kolmogorov-Chaitin complexity is a universal, one-tape Turing machine over the tape alphabet $\Gamma = \{0, 1, b\}$, with binary input and output.

As we have seen, this causes a number of problems because it is difficult to decode an input string of the form

$$p = q\, z$$

into an instruction part q and a data part z, with the intended semantics: run program q on input z.

Of course, this kind of problem would not surface if we used real programs instead of just binary strings. We should try to eliminate it in our setting, too.

The Book 4

M. Li, P. Vitányi
An Introduction to Kolmogorov Complexity and its Applications
Springer, 1993

Encyclopedic treatment.

But note that some things don't quite type-check ($\mathbb{N}$ versus $\mathbf{2}^\star$).

Terminology Warning 5

In language theory, a language $L \subseteq \Sigma^\star$ is said to be prefix if no string in L is a prefix of another.

Yes, this should be called prefix-free (some authors prefer this saner version), but the traditional form is prefix. Grin and bear it.

Always remember, obscure terminology keeps the uninitiated at bay. Works great in job interviews.

Prefix Programs

The key idea is to restrict our universal machine a little bit.

We require that $P \subseteq \mathbf{2}^\star$, the collection of all syntactically correct programs for $\mathcal{U}_{\text{pre}}$, is a prefix set: no valid program is a proper prefix of another.

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

Prefix Machines

Call a Turing machine $\mathcal{M}$ prefix if its halting set $\{\, p \in \mathbf{2}^\star \mid \mathcal{M}(p)\downarrow \,\}$ is prefix.

In order to compute $\mathcal{U}(p)$, machine $\mathcal{U}$ tries to parse the string p from left to right. Think about the machine tracing a branch in the tree $\mathbf{2}^\star$. If p turns out to be syntactically correct, it is executed. If not, $\mathcal{U}$ diverges.

There is never any ambiguity whether $p = q\,z$ or $p = q'\,z'$ as in the program/data approach.

Simulations

Simulations are also particularly simple for prefix machines: to simulate $\mathcal{M}$ on $\mathcal{M}'$ we can set up a header h such that

$$\mathcal{M}'(hp) \simeq \mathcal{M}(p)$$

for all $\mathcal{M}$-admissible programs p: $\mathcal{M}'$ can parse h, and then run $\mathcal{M}$ on the remaining string p.

This means that the difference in program-size complexity between various universal machines is rather small.

Converting to Prefix

We have to make sure that prefix machines exist and can do the same as ordinary ones.

Lemma

For any Turing machine $\mathcal{M}$, we can effectively construct a prefix Turing machine $\mathcal{M}'$ such that

- $\forall\, p \in \mathbf{2}^\star \left(\mathcal{M}'(p)\downarrow \Rightarrow \mathcal{M}(p) \simeq \mathcal{M}'(p)\right)$
- *$\mathcal{M}$ prefix* $\Rightarrow \forall\, p \in \mathbf{2}^\star \left(\mathcal{M}(p) \simeq \mathcal{M}'(p)\right)$

Of course, in general $\mathcal{M}'$ will halt on fewer inputs and the two machines are by no means equivalent (just think what happens to a machine with support $0^\star$).

As a consequence, we can effectively enumerate all prefix functions $\{e\}_{\text{pre}}$ just as we can effectively enumerate ordinary computable functions.

Proof

Suppose we have an ordinary machine $\mathcal{M}$ and some input $p \in \mathbf{2}^\star$. $\mathcal{M}'$ computes on p as follows:

- Enumerate the support of $\mathcal{M}$ in some sequence $(q_i)_{i\geq 0}$.
- If $q_i = p$, return $\mathcal{M}(p)$.
- If q_i is a proper prefix of p, or conversely, diverge.

Item 1 works since the domain of convergence of $\mathcal{M}$ is semidecidable, and thus recursively enumerable. We use the order of the enumeration to resolve prefix conflicts.

It is easy to check that $\mathcal{M}'$ is prefix and will define the same function as $\mathcal{M}$, provided $\mathcal{M}$ itself is already prefix.

□

Universal Prefix Machines

Since the support in our construction typically shrinks (potentially by a lot), we have to make sure that there is a universal prefix machine. Programming languages suggest that should work, but we need make sure we can handle this for Turing machines.

No problem, really, we can control the syntax of correct input strings. Say, define $\mathcal{U}'$ so that it checks for inputs of the form

$$p = 0u_1 0u_2 \ldots 0u_r\,1$$

Thus, if the input has the form $(0\mathbf{2})^\star 1$, then $\mathcal{U}'$ computes $\mathcal{U}(u_1 \ldots u_r)$ where $\mathcal{U}$ is an ordinary universal machine.

Otherwise it simply diverges.

Prefix Complexity

Definition

Let $\mathcal{U}_{\text{pre}}$ be a universal prefix Turing machine. Define the prefix Kolmogorov-Chaitin complexity of a string x by

$$C(x) = \min\big(\, |p| \mid \mathcal{U}_{\text{pre}}(p) \simeq x \,\big)$$

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

Of course, $C(x)$ is again not computable, for essentially the same reasons.

Connection

The following mutual bounds are due to Robert Solovay:

$$C(x) \leq K(x) + K(K(x)) + O(K(K(K(x))))$$

$$K(x) \leq C(x) - C(C(x)) - O(C(C(C(x))))$$

This pins down the cost of dealing with prefix programs as opposed to arbitrary ones. So the difference between $K(x)$ and $C(x)$ is not too large, we expect the non-leading terms on the right-hand side to be fairly small.

Still, $C(x)$ is much better behaved than $K(x)$ in many ways.

Print x Revisited

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

> **print** "$x_1 x_2 \ldots x_n$"

certainly does the job: this programming language is prefix. Alas, we need to worry about prefix TMs.

We could use delimiters around x as in the informal code snippet above, but remember that our input and output alphabet is fixed to be $\mathbf{2} = \{0, 1\}$.

We could add symbols to our base alphabet, but that does not solve the problem.

Self-Delimiting Programs

In the absence of delimiters, we can return to our old idea of self-delimiting programs. Informally, we could write

> **print next** n **bits** $x_1 x_2 \ldots x_n$

In pseudo-code this is fine, the only place where the prefix property could be violated is in the x part, but n fixes this problem. We obtain a complexity of at most $n + \log n + c$.

Again, we need to deal with prefix Turing machines, and that leads straight to cumbersome technical coding details.

To obtain a reliable bound on $C(x)$, there is no way around actually spelling out the coding details, at least in some detail.

A Standard Prefix Code

Here is a simple way to satisfy the prefix condition: code a bit string $x \in \mathbf{2}^*$ as

$$E(x_1 \ldots x_n) = x_1 0\, x_2 0 \ \ldots \ x_{n-1} 0\, x_n 1$$

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 $x = x_2 x_2 \ldots x_n$ 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 to one's heart's content. Let

$$E_0(x) := E(x)$$
$$E_{i+1}(x) := E_i(|x|)\, x$$

It is not hard to show that all the E_i are prefix codes.

Here is an example, using the string "abc ... xyz"

0 a0b0c0d0e0f0g0h0i0j0k0l0m0n0o0p0q0r0s0t0u0v0w0x0y0z1
1 1010001001|abcdefghijklmnopqrstuvwxyz
2 100011|11010abcdefghijklmnopqrstuvwxyz
3 1011|10111010abcdefghijklmnopqrstuvwxyz
4 1001|1110111010abcdefghijklmnopqrstuvwxyz
5 1001|1011101111010abcdefghijklmnopqrstuvwxyz

Here E_1 is optimal.

When To Stop?

But there is still a little problem: what is the optimal choice of k so that $E_k(x)$ has minimal length? Clearly k depends on the length of x; the longer x, the larger k.

For example, for a string of length $n = 10^{100}$, E_3 is optimal, it adds only 350 bits.

We can handle the cutoff problem nicely (for strings of length at least 2) by iterating the length function until we get to number requiring only 2 bits (i.e., 2 or 3). We define an "infinity code" E_∞ that keeps going until we get to that point.

Taking Things to the Limit

First define $\mathrm{len}^*(x)$, a discrete version of an iterated logarithm $\log^*$:

$$\mathrm{len}_1(x) = |x|$$
$$\mathrm{len}_{i+1}(x) = |\mathrm{len}_i(x)|$$
$$\mathrm{len}^*(x) = \min\big(\, i \mid \mathrm{len}_i(x) \le 2 \,\big)$$

Then we can use this iterated logarithm to determine the encoding level, for all strings x.

$$E_\infty(x) = E(k)E_k(x) \qquad \text{where } k = \mathrm{len}^*(x)$$

Example

Suppose we have bit-string x of length 20000.

We get the following values for $\mathrm{len}_i(x)$:

$$20000, 15, 4, 3$$

Written in binary these look like

$$100111000100000, 1111, 100, 11$$

So $\mathrm{len}^*(x) = 5$ and is encoded as a string of length $20000 + 15 + 4 + 3 + 2 * 2$, requiring 26 extra bits.

The E_∞ encoding looks like this:

$$100001 \;||\; 1011 \mid 100\,1111\,100111000100000\, x_1 \ldots x_{20000}$$

This could easily be improved a bit.

The Infinity Code

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

Lemma

$$|E_\infty(x)| = n + \log n + \log\log n + \log\log\log n \ldots + 2\log^*(n) + O(1)$$

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

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

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

print next n **bits** $x_1 x_2 \ldots x_n$

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. This is useful e.g. for concatenation.

Suppose we have prefix programs p and q that produce x and y, respectively.

But then pq is uniquely parsable, and we can easily find a header program h such that

$$h\, p\, q$$

is an admissible program for $\mathcal{U}_{\mathrm{pre}}$ that executes p and q to obtain xy.

Thus

$$C(xy) \le C(x) + C(y) + O(1)$$

Even Better

Define $C(x, y)$ to be the length of the shortest program that writes $x\,b\,y$ on the tape (recall that our tape alphabet is $\{0, 1, b\}$).

Note that $C(xy) \leq C(x, y) + O(1)$, but the opposite direction is tricky (think about $x, y \in 0^\star$).

At any rate, the last argument shows that $C()$ is subadditive:

$$C(x, y) \leq C(x) + C(y) + O(1)$$

This simply fails for plain KC complexity.

Better Mousetrap

From a more axiomatic point of view, plain KC complexity is slightly deficient in several ways:

- Not subadditive: $K(x, y) \leq K(x) + K(x) + c$.
- Not prefix monotonic: $K(x) \leq K(xy) + c$.
- Higher level complaint: plain KC complexity does not help much when applied to the problem of infinite random sequences.

Many arguments still work out fine, but there is a sense that the theory could be improved.

Here is the killer app for prefix complexity.

Chaitin's Ω

Definition

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

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

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

Lemma (Kraft Inequality)

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

But Why?

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

$$P(x) = \sum_{\mathcal{U}_{\text{pre}}(p) \simeq x} 2^{-|p|}$$

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

Then $\Omega = P(\mathbf{2}^\star)$. Ω depends quite heavily on the choice of $\mathcal{U}_{\text{pre}}$, so one could write $\Omega(\mathcal{U}_{\text{pre}})$ or some such for emphasis.

Proposition

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

In fact, for one particular $\mathcal{U}_{\text{pre}}$, one can show with quite some pain that

$$0.00106502 < \Omega(\mathcal{U}_{\text{pre}}) < 0.217643$$

Quoi?

To establish a result like $0.00106502 < \Omega(\mathcal{U}_{\text{pre}}) < 0.217643$ one needs to get down into the weeds and, following the details of the definition of $\mathcal{U}_{\text{pre}}$, show that

Lower Bound: show that some specific, short programs actually converge.

Upper Bound: show that some specific, short programs actually diverge.

So, this is unpleasantly close to Halting and rather messy in actuality–recall the Busy Beaver problem. For slightly longer programs, this type of analysis becomes quickly unmanageable.

Randomness

Proposition

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

Here $\Omega([n])$ denotes the first n bits of Ω. As a consequence, 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 quite poorly defined.

Halting and Ω

Lemma

Consider $q \in \mathbf{2}^n$. Given $\Omega[n]$, it is decidable whether $\mathcal{U}_{\mathrm{pre}}$ halts on input q.

Proof.

Start with a lower bound $\Omega_0 = 0$.

Dovetail computations of $\mathcal{U}_{\mathrm{pre}}$ on all inputs using the standard compute-things-in-stages approach.

Whenever convergence occurs on some program p, update the approximation: $\Omega_0 = \Omega_0 + 2^{-|p|}$.

Stop as soon as $\Omega_0 \geq \Omega[n]$. Then we have the following lower and upper bounds:

$$\Omega[n] \leq \Omega_0 < \Omega < \Omega[n] + 2^{-n}.$$

But then no program of length at most 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 less than 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.

The Abyss of Time

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

In fact, things are much, much worse than that.

Suppose some demon gave you these bits. 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.

All the answers would be staring at us, but we could not pull them out.

Hilbert's Dream

David Hilbert wanted to crown 2000+ years of development in math by constructing an axiomatic system that is

- consistent
- complete
- decidable

Alas . . .

Harsh Reality

Theorem (Gödel 1931)

Every consistent reasonable theory of mathematics is incomplete.

Theorem (Turing 1936)

Every consistent reasonable theory of mathematics is undecidable.

Reasonable here just means: at least as strong as basic arithmetic, and the axioms are decidable.

This is good news for anyone interested in foundations, who would want to live in a boring world?

The Proofs

Gödel's argument is a very careful elaboration and formalization of the old Liar's Paradox:

> This here sentence is false.

Turing uses classical Cantor-style diagonalization applied to computable reals.

Both arguments are perfectly correct, but they seem a bit ephemeral; they don't quite have the devastating bite one might expect.

Incompleteness

For example, Gödel's version of the Liar is an arithmetic statement φ that says: this sentence is not provable in the given system.

By consistency, φ must indeed fail to be provable in the chosen system, hence the sentence is true—so our theory is incomplete.

The problem is that the sentence in question is logical in nature, rather than just pure arithmetic. A lot of work has since gone into finding true but unprovable statements that are more mathematical in nature. Still, they are not totally compelling. Look at Harvey Friedman's work for several examples.

Ω can help to make the limitations of the formalist/axiomatic approach much more concrete. First a warm-up.

Normal Numbers

Émile Borel defined a normal number in base B to be a real r with the property that all digits in the base B expansion of r appear with limiting frequency $1/B$.

Theorem (Borel)

With probability 1, a randomly chosen real is normal in any base.

Alright, but how about concrete examples? It seems that $\sqrt{2}$, π and e are normal (billions of digits have been computed), but no one currently has a proof.

Champernowne's Number

$$C = 0.12345678910111213141516171819202122\ldots$$

Champernowne showed that this number is normal in base 10 (and powers thereof), the proof is not difficult.

Proposition

Ω is normal in any base.

Of course, there is a trade-off: we don't know much about the individual digits of Ω.

Proving Theorems

Time to get serious. Fix some cutoff n. Suppose we want to prove all correct "theorems" of the form

$$C(x) = m < n \qquad \text{or} \qquad C(x) \geq n$$

In other words, we want to prove a lower bound N for some concrete string x, or pin down the complexity exactly.

How much information would we need to do this? After all, we need to figure out convergence of our universal prefix machine, a non-trivial matter.

What we need is the maximal halting time τ of all programs of length at most $n-1$. It is not hard to see that

$$C(\tau) = n + O(1)$$

Essentially, nothing less will do. But this is the computational approach, we want to reason about theorem proving.

Theories

In the following we assume that $\mathcal{T}$ is some axiomatic theory of mathematics that includes arithmetic (first-order logic is fine).

Think of $\mathcal{T}$ as Peano Arithmetic, though stronger systems such as Zermelo-Fraenkel with Choice is perfectly fine, too (some technical details get a bit more complicated; we have to interpret arithmetic within the stronger theory).

Assertions like $C(x) = m$ and $C(x) \geq n$ can certainly be formalized in $\mathcal{T}$ and we can try to determine how easily these "theorems" might be provable in $\mathcal{T}$.

Consistency

We need $\mathcal{T}$ to be consistent: it must not prove wrong assertions. This is strictly analogous to the situation in Gödel's theorem: inconsistent theories have no trouble proving any assertion whatsoever.

Actually, technically all we need is Σ_1 consistency: any theorem of the following simple form, provable in $\mathcal{T}$, must be true:

$$\exists\, x\, \varphi(x)$$

where φ is primitive recursive (defines a primitive recursive property in $\mathcal{T}$) and the existential quantifier is arithmetic.

Measuring Theories

We assume that suitable rules of inference for first-order logic are fixed, once and for all. So the theory $\mathcal{T}$ comes down to its set of axioms.

If there only finitely many axioms, we can think of their conjunction as a single string and define $C(\mathcal{T})$ accordingly.

If there are infinitely many axioms (as in PA, think about induction), the set of all axioms is still decidable and we can define $C(\mathcal{T})$ as the complexity of the corresponding decision algorithm.

Note that this approach totally clobbers anything resembling semantics: it does not matter how clever and/or elegant the axioms are, just how large a data structure is needed to specify them.

Chaitin's Theorem

Theorem (Chaitin 1974/75)

If $\mathcal{T}$ proves the assertion $C(x) \geq n$, then $n \leq C(\mathcal{T}) + O(1)$.

Proof.

Enumerate all theorems of $\mathcal{T}$, looking for statements of the form $C(x) \geq n$.

For any $m \geq 0$, let x_m be the first theorem so discovered where $n > C(\mathcal{T}) + m$.

By consistency, we have

$$C(\mathcal{T}) + m < C(x_m)$$

By construction,

$$\begin{aligned} C(x_m) &\leq C(\mathcal{T}, C(\mathcal{T}), m) + O(1) \\ &\leq C(\mathcal{T}) + C(m) + O(1) \end{aligned}$$

Proofs are Useless

This is one place where subadditivity is critical.

But then it follows that

$$m < C(m) + O(1)$$

and thus $m \leq m_0$ for some fixed m_0.

□

Similarly one can prove that no consistent theory can determine more than

$$C(\mathcal{T}) + O(1)$$

bits of Ω.

We have a perfectly well-defined real, but we can only figure out a few of its digits.

Number Theory

Here is an application of Ω in number theory. Recall

Theorem (Y. Matiyasevic, 1970)

It is undecidable whether a Diophantine equation has a solution in the integers.

One important step towards the proof was to show that any semidecidable set can be expressed in terms of a exponential Diophantine equation:

$$a \in A \iff \exists\, x_1, \ldots, x_n\ E(a, x_1, x_2, \ldots, x_n) = 0$$

Of course, exponential Diophantine equations are scary in general, even over $\mathbb{N}$:

$$(x+1)^{n+3} + (y+1)^{n+3} = (z+1)^{n+3}$$

Counting Roots

Incidentally, if $E(a, x_1, x_2, \ldots, x_n) = 0$ has α solutions, then there is a program of size

$$C(\alpha) + O(1)$$

to find them: brute-force search until they are all discovered.

But how about infinitely many solutions?

Theorem (Chaitin)

There is an exponential Diophantine equation $E(k, \boldsymbol{x}) = 0$ such that: there are infinitely many solutions $\boldsymbol{x}$ iff the kth bit of Ω is 1.

Loosely speaking: randomness and chaos lurks even within integer polynomials–not a place where one would usually be looking for these scary things.

Solovay's Theorem

One can sharpen Chaitin's theorem to a point where it almost seems absurd:

Theorem

Let $\mathcal{T}$ be as before. Then there is a universal prefix machine $\mathcal{U}_{\mathrm{pre}}$ such that

- *Peano Arithmetic proves that $\mathcal{U}_{\mathrm{pre}}$ is indeed universal.*
- *$\mathcal{T}$ cannot determine a single digit of $\Omega = \Omega(\mathcal{U}_{\mathrm{pre}})$.*

This result is rather counter-intuitive, one might think that the standard approach towards identifying a few digits of Ω should work just fine. The proof depends on a very clever construction of a particular universal prefix machine and uses Kleene's recursion theorem.

Proof Sketch

Start with any old universal prefix machine $\mathcal{V}$ whose universality can be proven in PA (any standard machine will do).

Define $\mathcal{U}_{\mathrm{pre}}$ in three cases as follows:

- $\mathcal{U}_{\mathrm{pre}}(\varepsilon) \uparrow$
- $\mathcal{U}_{\mathrm{pre}}(0p) \simeq \mathcal{V}(p)$
- $\mathcal{U}_{\mathrm{pre}}(1p)$: batten down the hatches

Note that $\mathcal{U}_{\mathrm{pre}}$ is already guaranteed to be universal, provably in PA. We will use the missing inputs $1p$ to destroy any semblance of predictability of Ω.

Kleene to the Rescue

Let $|p| = n$. We will not use p as an actual program (unlike in the $0p$ case) but only as a yardstick to determine n, and later to determine a rational number.

Enumerate the theorems of $\mathcal{T}$ of the form

$$\Theta(n, b) = \text{the } n\text{th bit of } \Omega \text{ is } b.$$

Here $b \in \mathbf{2}$ is a single bit.

Wait, we are in the middle of the definition of $\mathcal{U}_{\mathrm{pre}}$, but the formalization of $\Theta(n, b)$ requires knowledge of $\mathcal{U}_{\mathrm{pre}}$. This is when the Recursion Theorem strikes: in the definition of $\mathcal{U}_{\mathrm{pre}}$, we may safely assume that we have access to an index for $\mathcal{U}_{\mathrm{pre}}$.

We ignore all messy details.

What's Wrong?

Suppose we discover $\Theta(n, b)$. Now do the following:

For any program p of length n, let r_0 be the rational with dyadic expansion $pb \in \mathbf{2}^{n+1}$ and let $r_1 = r_0 + 2^{-(n+1)}$. Note that for the right p we must have $r_0 < \Omega < r_1$.

Now search for a stage $\sigma \geq 0$ where we already have

$$r_0 < \Omega_\sigma < r_1$$

for the standard, computable approximation Ω_σ. If the search succeeds, let $\mathcal{U}_{\mathrm{pre}}(1p) = \varepsilon$, adding a bit to Ω and kicking us out of the interval.

But if $\mathcal{T}$ correctly determines the nth bit of Ω (proves the right $\Theta(n, b)$), then Ω will be in this interval, for the right program p of length n.

Contradiction.