An Introduction to Algorithmic Information Theory

Its history and some examples

GEORGE MARKOWSKY

OVERVIEW

The goal of this paper is to provide a simple introduction to Algorithmic Information Theory (AIT) that will highlight some of the main ideas without presenting too many details. More technical treatments of these ideas can be found in References [1], [2], [3] and [4], which are listed at the end of the paper. The main ideas of Algorithmic Information Theory will be presented using English as the underlying programming language. The presentation illustrates the fact that the same arguments can be expressed in any other reasonable language and that the main results have a robust universality across all reasonable languages.

This paper grew out of a short course on AIT that Gregory Chaitin presented in June 1994, at the University of Maine. I helped with the course and observed some of the topics that proved most difficult for students. I presented a series of lectures based on these observations at the 1995 Summer School on Algorithmic Information Theory held in Mangalia Romania. The text of those lectures, and others from that workshop, can be found in *The Journal of Universal Computer Science.*[8] All the material presented here is based on the work of Gregory Chaitin.

The following topics will be discussed in this paper.

- **What is AIT?**
- **Some important pre-AIT ideas**
- **Fundamental Concepts of AIT**
- **The Uncomputability of Algorithmic Complexity**
- **The Hacker's Problem**
- **The Halting Problem**
- **Chaitin's Ω and Ω^{--}**
- **Properties of Ω**
- **Some Implications of AIT**

George Markowsky is with the Computer Science Department at the University of Maine, Orono, ME.

WHAT IS AIT?

AIT, of course, stands for Algorithmic Information Theory. The *information* part of the name comes from Shannon's information theory, which first proposed measuring the amount of information. The *algorithmic* part of the name comes from the fact that algorithms (programs) are used for measuring information content.

SOME IMPORTANT PRE-AIT IDEAS

The Concept of an Algorithm

A very interesting discussion of this is found in Volume 1 of Knuth's *The Art of Computer Programming.* After dismissing several earlier derivations, Knuth presents what he claims is the correct derivation. He notes that even as late as 1957, algorithm did not appear in *Webster's New World Dictionary.*

The closest word to appear in dictionaries of that time was *algorism,* which means the process of doing arithmetic using Arabic numerals. The word *algorithm* appeared as the result of confusing the word arithmetic with the name of the Persian mathematician Abu Jáfar Mohammed ibn Mûsâ al-Khowârizmî (c. 825). The word was probably first widely used for Euclid's GCD algorithm before it came to have its present meaning of a well-defined procedure for computing something. The 1971 edition of *The Compact Edition of the Oxford English Dictionary* gives the following definition for algorithm: "erroneous refashioning of algorism."

The History of Entropy

Nicolas Leonard Sadi Carnot (1796-1832) was an engineer who was interested in understanding how much work can be produced by heat engines such as steam engines. He introduced many important concepts into thermodynamics including those of reversible and irreversible engines.

Building upon his work, Rudolf Clausius introduced the concept of *entropy*:

CCC 1076-2787/97/04014-09

"I propose to name the magnitude S the entropy of the body from the Greek word 'η τροπη, a transformation. I have intentionally formed the word entropy so as to be as similar as possible to the word *energy*, since both these quantities, which are to be known by these names, as so nearly related to each other in their physical significance that a certain similarity in their names seemed to me advantageous..."

Further contributions to the field were made by Kelvin, Boltzmann, and Gibbs. The last two connected entropy to statistical mechanics and showed that it can be viewed as a measure of the *randomness* of a collection of entities such as molecules.

The work of Boltzmann and Gibbs lead to the following expression for entropy: $-c\Sigma_i p_i log p_i$ where c is some constant > 0 and the p_i are probabilities of various states.

Information Theory

The work of Claude Shannon shows the connection between the entropy of thermodynamics and the entropy of information theory. Many of the ideas of information theory were in the air when Claude Shannon wrote his two classic papers on information theory in 1948. Shannon has shown that it makes sense to measure the information content of messages and how to transmit messages correctly in the presence of noise. The importance of this work cannot be overestimated.

Shannon derived the connection between the entropy of thermodynamics and the information-theoretic entropy that he used in his theory. It is interesting to note that information-theoretic entropy is the unique (up to multiplicative constant) continuous function, F, on probability vectors (vectors of nonnegative numbers that sum to 1) that satisfies the following three very simple conditions.

- For a given n, F achieves its maximum over all probability vectors of length n at $(1/n,\ldots,1/n)$.
- Extending a probability vector by adding an additional entry of 0 to its end does not change the value of F.
- Let A and B be two probability vectors and C be the probability vector that represents the probabilities of the joint events from A and B happening. Then $F(C) = F(A) + F_A(B)$, where $F_A(B)$ represents a conditional computation to be performed using the values of B and the knowledge of events in A.

For more details consult Khinchin's book. [6]

The Berry Paradox

In the *Principia Mathematica*, Russell and Whitehead describe an interesting paradox that lies at the heart of AIT and provides the key insight into many of its results. To start off, first note that there are only finitely many English phrases not exceeding a certain length in words. Some of these phrases will define positive integers unambiguously. Make a list of these phrases of twenty words or less. (Alternatively, you can also limit the number of characters.)

This list will define a finite set of integers, so some positive integers will not be on the list. Let Q be the smallest positive integer not on the list. Now consider the following phrase: "the smallest positive integer that cannot be defined in less than twenty words." This is a phrase of 13 words!

Undecidability

Hilbert was of the opinion that eventually algorithms would be found to solve all outstanding problems in mathematics and some of his famous problems presented at the International Mathematical Congress of 1900 tried to direct research at finding these algorithms. In 1931 Kurt Gödel published a paper showing that any system of mathematics complicated enough to include arithmetic must contain problems undecidable from the basic axioms in that system.

In 1937 (a year before he earned his Ph.D.), Turing published the paper in which he introduced "Turing machines" and showed that the Halting Problem, deciding whether a Turing machine running on a particular input would halt or not, is undecidable. Turing's work is of critical importance in computer science and significantly simplified the demonstration that there are undecidable problems in formal systems.

Credit for AIT Discoveries

Three people are generally credited with codiscovering some of the basic ideas of AIT: Chaitin, Kolmogorov, and Solomonoff. Solomonoff published some definitions before Chaitin and Kolmogorov. Chaitin has contributed the largest body of work to AIT and continues to extend the field at the present time. There is a tendency on the part of some writers to attach Kolmogorov's name to everything in the field. This makes little sense and is obviously unfair to other researchers.

A curious justification for this is stated in Li and Vitányi's book [7, p. 84]

> This partly motivates our choice of associating Solomonoff's name with the universal distribution and Kolmogorov's name with algorithmic complexity, and, by extension, with the entire area. (Associating Kolmogorov's name with the complexity may also be an example of the "Matthew Effect" first noted in the Gospel according to Matthew, 25:29-30, "For to every one who has more will be given, and he will have in abundance; but from him who has not, even what he has will be taken away. And cast the worthless servant into the outer darkness; there men will weep and gnash their teeth.")

SOME FUNDAMENTAL CONCEPTS OF AIT

I will loosely define the *algorithmic complexity of a finite string* as the *size* of the smallest program that generates that particular character string. More generally, one can define the complexity of any object as the size of the smallest program that generates it. These definitions are loose because they leave out some key details:

- What language are we using?
- What does generate mean?
- How do we measure program size?
- What exactly constitutes a program?
- How do we handle infinite strings?

For best results, a programming language should be used because of the great precision of programming languages. We can generally tell what constitutes a legal program in a particular programming language. Human languages, on the other hand, are full of ambiguous statements and it is not always clear whether a legal and unambiguous description has been given. Nevertheless, since our goal is to describe the main ideas of AIT we will do so using English as our programming language. The references indicate where more technical treatments of AIT can be found.

If we use English, our definition of the algorithmic complexity of a finite character string reduces to "the *smallest* number of characters that it takes to describe in English how to generate a particular string." We count all characters including blanks and punctuation to come up with a number. I will use the term *program* to refer to an English description of a string. It is important to note that the word smallest is emphasized in the text above. Many key results depend on the fact that the complexity is the smallest possible description.

An important point to stress here, and one to which I will return, is that the algorithmic complexity of an object depends very much on the language in which the object is described! We can make the complexity of any particular object as small or as large as we choose by picking the appropriate language or by modifying an existing language.

For example, AIT is not a traditional English word. We have added it to English language as an abbreviation for Algorithmic Information Theory. If one of the rules of English is to automatically expand abbreviations then AIT is a quick way to generate a much larger string. Similarly, many languages permit the addition of abbreviations, so you can create a new language, consisting of the old language with an abbreviation, in which the size of the particular string would be reduced.

This feature is not all that different from other forms of measurement. For example, it is incorrect to say that the length of a particular object is 4.5 without specifying the scale. You will get very different "lengths" depending on whether you use feet, meters or light years as your basic unit.

The key point here is to make evident the universal features of AIT—those that hold independent of the language used to compute complexity. In general, by choosing the language appropriately we can alter the results for some finite collection of strings, but we cannot alter the results for the totality of strings.

Roughly speaking, the algorithmic complexity of a finite string always exists because there are only finitely many English descriptions having a size less than or equal to a given number, and because every finite string can be generated by a program that essentially just outputs the string.

Knowing that the algorithmic complexity of a string exists and finding it are two different things. There are other sorts of complexity and even variations of Algorithmic Complexity. Since these notes are an introduction to AIT, I will just present one version of complexity.

It is important to realize that the complexity of most strings cannot be substantially shorter than the string itself. To make this statement more precise, let's assume that English has an alphabet with Q letters. Q depends on the punctuation marks that we count, along with other characters such as blanks, tabs, carriage returns, etc. The results do not depend on the precise value of Q, so we will present this argument in general.

Once we agree on the value of Q, we note that there are Q^n strings of length n. Suppose that we assume that to each "program" (description) we associate at most one string that it produces. It is possible that no string is produced if the program goes into an infinite loop or suffers from some ambiguity. Since programs are also strings, it is easy to see that the number of programs having length $<n$ is bounded above by

$$Q^0 + Q^1 + \ldots + Q^{n-1} = (Q^n - 1)/(Q - 1) < Q^n/(Q - 1) .$$

Thus, at most, $1/(Q - 1)$ of all strings of length n can be expressed by programs that are shorter. Thus, at least $(Q-2)/(Q-1)$ of all strings of length n require a program of length at least n to generate. I will use $H(S)$ to denote the *algorithmic complexity* or *program-size complexity* of S. The preceding discussion has the following consequence.

Theorem 1: (Arbitrary Complexity Theorem) Given any integer, N, there are strings with algorithmic complexity greater than N.

SOME EXAMPLES

Here are some simple examples that will help you to better understand how AIT works. Let's start with a very simple example. Suppose you are given the string "asbdasdferrasdfa." One way to describe it in English is to simply say *the string is asbdasdferrasdfa* which is only 14 characters longer than the string we wish to describe. Thus, the English algorithmic complexity of a string cannot exceed its length by more than 14. Thus, we have our first theorem.

Theorem 2: (The Upper Bound Theorem) For any string, S,

$$H(S) \leq |S| + 14 .$$

The Upper Bound Theorem is not quite correct as stated, because we have glossed over a very important point: how can we tell when strings begin and end? At first glance, one might think that it's possible to use some special character to mark the end of the string, but since this character is part of the alphabet we must also provide a mechanism for including this character as part of a string if we want to. There are basically two different approaches to this problem. One is to use an *escape character* of some sort that permits you to indicate precisely which characters are part of a string and which are not. The other approach is to use a length indicator of some sort. These approaches are discussed in the references and indicate some of the technical complexities that must be dealt with if one wants to get the theory completely correct. Depending on the approach taken, the correct form of the Upper Bound Theorem might read something like:

$$H(S) \leq 2|S| + C$$

or

$$H(S) \leq |S| + log(|S|) + C$$

where the constants may be more precisely specified. We do not need to know the various constants precisely. For our purposes, it is enough to state the Upper Bound Theorem as "the algorithmic complexity of a string cannot be significantly larger than the length of the string."

One last complication needs to be mentioned: long strings are generally not written on a single line, but require carriage returns and line feeds to break the string into lines. It seems reasonable that we should count these extra characters. If we do, this will modify the Upper Bound Theorem even more, but the large picture will remain the same.

For other languages, the Upper Bound Theorem remains essentially the same. The constants will be different, but the implications are language independent.

For many strings, S, $H(S)$ is significantly smaller than the bound provided by the Upper Bound Theorem. For example, consider the following string:

aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa

We can describe this string as 50 *a*'s which takes 6 characters instead of the 64 characters suggested by the Upper Bound Theorem. A little reflection shows how it is possible to generate extremely large strings using relatively short descriptions. For example, consider the following description.

Doubling a string S *means replacing* S *by* SS, *where* SS *means writing two copies of* S *side by side. Produce a string by doubling* S *1000 times starting with* S = "a."

The preceding description will produce a string of length roughly 10^{301}. This number is larger than the estimated number of atoms in the universe, so it would be quite difficult to write it out using standard numerical notation.

THE UNCOMPUTABILITY OF ALGORITHMIC COMPLEXITY

I hope that the preceding discussion has given you some insight into actually measuring H for various strings. By now, you are probably eager to start computing some values of H.

Unfortunately, I will now prove that this is more difficult than might appear. It turns out that it is impossible to compute H for more than a finite number of strings!

An argument based on the Berry Paradox shows quite convincingly that H is not computable for character strings. The argument is roughly the following. Assume that we have a fixed English description that explains how to correctly compute the complexity of strings. We can use this description to derive a description for computing of complexity greater than the size of the description, which is impossible. The details are given in the proof of the following theorem.

Theorem 3: (Uncomputability of H Theorem) No English description can correctly compute H for all finite strings.

Proof: This proof is by contradiction. Assume that we have an English description that computes H and consider the following English description.

1. Assign the empty string "" to S.
2. Generate the next string following S in size-alphabetical order and assign its value to S.
3. Compute the complexity of S.
4. If the complexity of S is $\leq N$ then go to Step 2.
5. Display S.

Before explaining the steps in detail, it is worth answering the following question: *what does this description produce?* Clearly this procedure will continue looping until a string of complexity $>N$ is produced. Note that so far I have not specified N. This was a deliberate omission since N has to be large enough for our purposes. The subsequent discussion will give you insight into the size of N. Let's review each step and most importantly its size!

The first step is fairly clear. Let's denote its size by C_1. C_1 depends on whether we count carriage returns and line feeds and how much overhead we wish to assign for the ordered list structure.

I want to clarify the nature of the second step. In particular, given two strings S and T, we will say that $S < T$ in size-alphabetical order if either the length of S is strictly less than the length of T or S and T have the same length but S precedes T alphabetically. You need to decide how much explanation is required by the second step. Let's denote the size of this step by C_2.

The third step is the one that we assume exists. In particular, we can denote the size of this step by C_3. I cannot say much more about this step since I want to show that no such description is possible. For the time being, we assume that this step is possible.

Step 4 seems simple, but measuring its size involves a subtle point. Since we did not specify N, we cannot have a fixed size for this step. In general, specifying N requires that we use $\log N$ characters to specify the number in the usual decimal notation. Thus, the size of this step is $C_4 + \log N$.

Step 5 is fairly simple and we assign it size C_5.

If we add up the sizes of the various steps, we see that the description above requires the following number of characters to specify:

$$C_1 + C_2 + C_3 + C_4 + \log N + C_5.$$

However, assuming that we know how to compute correctly $H(S)$, we are guaranteed to produce a string of complexity greater than N. Note that the definition of complexity implies that a string of complexity greater than N can be produced only by an English description of length greater than N. Thus, we get the following inequality

$$C_1 + C_2 + C_3 + C_4 + \log N + C_5 > N.$$

Note that only finitely many N can satisfy the above inequality, since by choosing N sufficiently large we can make N larger than the left side of the inequality. Since N can be chosen as desired, we choose a large enough N to violate the inequality and produce a contradiction. The fact that we have arrived at a contradiction, means that it is not possible to construct Step 3 (all the other steps have been constructed), which is what we wanted to prove!

PROPERTIES OF *H*

As we have seen, H is not computable. Nevertheless we can derive some of its properties. The following is an example of this.

Theorem 4: (Concatenation Theorem)

$$H(s_1 + s_2) \leq H(s_1) + H(s_2) + C$$

where + between strings indicates *concatenation*, which is the creation of a new string by joining the beginning of the second string to the end of the first.

Proof: Consider the following description: simply explain how to concatenate two strings and then give the descriptions of the two strings. The size of this description is the sum of the descriptions for the individual strings together with whatever it takes to describe how to concatenate two strings into one.

THE HACKER'S PROBLEM

Closely related to the problem of computing H is the *Hacker's Problem*: how can you show that a given English description is the shortest English description that can generate a particular string? Gregory Chaitin refers to such shortest descriptions as *elegant descriptions*.

To minimize problems, let's accept the following convention. The empty description, i.e., the description that has no words in it, produces the empty string. Clearly, the empty description is the elegant description that produces the empty string. The empty string can also be produced by any description that does not output any characters.

In some cases it is useful to have descriptions that produce an infinite sequence of characters. For example, the decimal description of the fraction 1/3 consists of an infinite sequence of 3's. Nevertheless, dealing with infinite strings is a tricky business since most of them cannot be generated by any language. To simplify our discussion, the only descriptions that will be considered elegant are those that eventually produce a finite string and halt.

I base the name, The Hacker's Problem, on the book *Hackers* by Stephen Levy which details how many hackers were on an endless quest to find the shortest program to accomplish some task. The hackers were never able to prove that some program was the shortest. We will now see why this is no accident. Gregory Chaitin participated in such a quest when he was learning programming while in high school.

In view of the uncomputability of H, you might suspect that the Hacker's Problem is also not solvable. We will now see how to prove this using the Berry Paradox.

Usually, when people talk about proving something they imply that one must use an axiomatic system of some type. The same object can be achieved by thinking of an axiomatic system as a computer program that can correctly decide a certain question. For our purposes, we can think of an axiomatic system as an English description.

The theorem below shows that it is impossible for any description to decide correctly for infinitely many descriptions that they are elegant descriptions. Furthermore, the size of the description being used to decide on the elegance of other descriptions bounds the complexity of the descriptions for which minimality can be demonstrated. Thus, the checking description cannot correctly decide whether descriptions that are significantly larger than itself are elegant!

Finally, I should note that an English description can be thought of as a character string. The structure of a description is indicated by special characters such as carriage returns and line feeds. In fact, word processing documents may be thought of as long character strings.

Theorem 5: (Hacker's Problem is Unsolvable Theorem) The Hacker's Problem cannot be solved in general, i.e., there are only *finitely* many provably elegant programs.

Proof: This proof proceeds by contradiction. Suppose that there was an English description that could correctly determine whether any other English description was elegant. Consider the following description.

1. Assign the empty string "" to S.
2. Generate the next string following S in size-alphabetical order and assign its value to S.
3. Determine if the string S is a correct English description. If not, go to Step 2.
4. If the size of S is $\leq N$, then go to Step 2.
5. If the description is not elegant, then go to Step 2.
6. Generate the string that is defined by the expression.

Before I describe the stages in more detail, consider what the output of this description must be. If we choose N greater than 0, then it is clear that the description above must produce a string of complexity $>N$. As in the proof of Theorem 3, I will derive a contradiction by picking N large enough.

Steps 1 and 2 are the same as considered in the proof of the uncomputability of H. As before, we can give their sizes as C_1 and C_2.

Step 3 takes some explanation. The idea here is to determine a string generated in Step 2 is a legal English description. You might consider this to be equivalent to providing a book of grammar. This step is a bit clearer when dealing with a computer language, since just about every language has a *compiler* associated with it. A compiler is a program that tells whether a computer program is actually a correct program in the language in which it purportedly is written. Let's use C_3 to represent the size of this step. C_3 can be quite large, but the important thing is that it is a constant.

As in proving the uncomputability of H, the size of Step 4 is $C_4 + \log N$.

We are assuming that Step 5 is possible, and that it requires a description of size C_5. We are also using the convention that descriptions which do not halt are not elegant, so if a description is declared elegant it will halt and produce an output string.

Finally, Step 6 has some finite size C_6.

Since the final output of this description is a string of complexity $>N$, it follows that the size of this description must also be $>N$. This gives the following inequality

$$C_1 + C_2 + C_3 + C_4 + \log N + C_5 + C_6 > N,$$

which is impossible.

We can draw some additional information from this argument. In particular, we have that

$$C_5 > N - (C_1 + C_2 + C_3 + C_4 + \log N + C_6).$$

Recall that C_5 was the size of the elegance-recognition description. The conclusion here is that any description that can prove that a description of size N is elegant, essentially must have size at least N itself. Thus, descriptions cannot prove the elegance of descriptions that are substantially larger than they are themselves.

THE HALTING PROBLEM

The Halting Problem was mentioned earlier in connection with Turing machines. You can formulate a variant of the Halting Problem in any computational scheme that you choose. In particular, the Halting Problem for English descriptions is deciding whether a particular English description of a string produces a finite string or whether it runs forever. As with most general questions about programs (descriptions), the Halting Problem is not solvable. In practical terms, our inability to solve the Halting Problem and related problems means that we cannot come up with foolproof tests for viruses or bugs.

Many people are surprised to learn that the Halting Problem cannot be solved. It seems difficult to accept that it is not obvious when a procedure will not halt.

I will now show how our inability to compute $H(S)$ or to solve the Hacker's Problem implies that the Halting Problem is not solvable. In both cases, the argument is almost the same.

First, let's show that if the Halting Problem is solvable by a description, we could construct a description that could compute $H(S)$. Since we know that the latter is impossible, it follows that solving the Halting Problem is also impossible. We summarize this result in the following Theorem.

Theorem 6: (Computing H from the Halting Problem Theorem) If there were a description that could tell whether any description halted or not, then there would be a description that could compute H.

Proof: As before, we assume that there is a description that solves the Halting Problem. Now consider the following description which will compute $H(S)$ for any nonempty string S. For the empty string, we can use some convention such as assuming that the empty description produces the empty string.

1. Record S.
2. Assign the empty string "" to T.
3. Generate the next string following T in size-alphabetical order and assign its value to T.
4. Determine if the string T is a correct English description. If not, go to Step 3.
5. Determine whether T halts. If not, go to Step 3.
6. Carry out the instructions in the description T to produce a string U.
7. If $U = S$, then the length of T is $H(S)$. If $U \neq S$, go to Step 3.

The above description computes $H(S)$ because it goes through the descriptions in size order. Thus, the first time that

it finds a description that produces S, we are sure that we have found an elegant description that produces S, and hence the length of this description is the complexity of S. Note that being able to solve the Halting Problem keeps us from wasting time by trying to compute with descriptions that never halt.

Theorem 7: (Unsolvability of the Halting Problem Theorem) The Halting Problem is unsolvable.

Proof: From Theorem 6, it follows that if the halting problem were solvable, we could compute $H(S)$ for all S. However, we know that $H(S)$ cannot be computed, so therefore the Halting Problem cannot be solved.

We could also prove that the Halting Problem is unsolvable by using the Hacker's Problem as our basic unsolvable problem. We would first show that being able to solve the Halting Problem would enable us to solve the Hacker's Problem and then we would conclude, as in Theorem 7, that since the Hacker's Problem is unsolvable, so is the Halting Problem. The argument that being able to solve the Halting Problem enables us to solve the Hacker's Problem is given below.

Theorem 8: (Solving the Hacker's Problem from the Halting Problem Theorem). If we could solve the Halting Problem, we could solve the Hacker's Problem.

Proof: As in the proof of Theorem 6, we assume that we have a description that permits us to decide whether an English description halts or not. Now consider the following description which will decide whether a description that halts is elegant. Let's assume that the description that we want to analyze is given by the string U.

1. If U does not halt, stop the process and declare it to be not elegant.
2. Compute the string described by U and store it in S.
3. Assign the empty string "" to T.
4. Generate the next string following T in size-alphabetical order and assign its value to T.
5. If the length of T is ≥ the length of U, output the result that U is elegant and stop.
6. Determine if the string T is a correct English description. If not, go to Step 4.
7. Determine whether T halts. If not, go to Step 4.
8. Carry out the instructions in T to produce a string W.
9. If $W = S$, then output the statement that U is not elegant and stop. Otherwise, go to Step 4.

The previous argument can be summed up as follows. First, decide whether the description halts or not. If it does not halt, it is not elegant using our conventions. Otherwise, go through all possible shorter English descriptions. Use our hypothesized ability to solve the Halting Problem to skip over the English descriptions that do not halt and carry out the ones that halt. If any of these shorter descriptions produce the same string as the description we are given, then that description is not elegant. Otherwise, it is elegant.

Chaitin et al. [5] contains two proofs that show that being able to compute program-size complexity would confer the ability to solve the Halting Problem.

CHAITIN'S Ω AND Ω^{--}

Gregory Chaitin has found an interesting way to combine his incompleteness results into one neat number which he calls Omega (Ω). The bits of Ω are truly unknowable. To correctly define Ω we need to introduce quite a bit of machinery. Since I want to keep this exposition simple, I will define a number Ω^{--} which is much easier to define and which captures much of the flavor of Ω. For details on defining Ω, see [1, 4, 8].

We have already described how descriptions can be thought of as a single string if we include special characters such as carriage returns and line feeds in our alphabet. We have already discussed how to order strings in size-alphabetical order. Now imagine that all descriptions are ordered in size-alphabetical order and written one after the other in a list as shown below.

Description 1
Description 2
Description 3
...

Imagine further that each description is associated with the digit 0 if it does not halt and the digit 1 if it halts. In particular, we might have the following diagram.

0 or 1	Description 1
0 or 1	Description 2
0 or 1	Description 3
...	...

Reading the first column of the preceding table we can make a decimal number that looks like $.d_1d_2d_3...$ where each d_i is 0 or 1 depending on whether the ith description halts or not. Let's call this number Ω^{--}. Ω^{--} has many of the properties of Ω, but is not as densely packed. It avoids some of the technical difficulties that arise when Ω is defined. For more details on defining Ω, see [1-5, 8]. Chaitin calls Ω the *Halting Probability*.

Since the definition of Ω^{--} is bound up with knowing which programs halt it should not be surprising to learn that Ω^{--} is very uncomputable. It is clear that Ω^{--} exists and is a number between 0 and 1/9 = .11111....

If you could know the digits of Ω^{--} you would know a lot! In particular, if you knew a particular bit of Ω^{--} you could tell whether the corresponding program halted or not. Ω would also give you this information, but since it is a compressed form of Ω^{--} it is more involved to figure out whether a particular program halts using its digits.

The above observation permits us to show that Ω^{--} is not computable in the sense that there is no description that will generate the digits of Ω^{--} sequentially. So far we have concentrated on descriptions that produce a finite string and halt. It also makes sense to envision a description that never halts, but continues to output an infinite sequence of digits. For example, one description of the number 1/3 is .3333..., which might be given by the following description:

1. Write down "."
2. Write a "3" at the end of the current string.
3. Repeat Step 2.

It is clear that every rational number has a description like the preceding one. Let's call every number that has such a description *computable*. Many irrational numbers such as $\sqrt{2}$, π, and e are computable since we can set up descriptions that would generate all the digits of the number in sequence. We give one important property of Ω^{--} in Theorem 9.

Theorem 9: (The Uncomputability of Ω^{--} Theorem) Ω^{--} is not a computable real number. In particular, Ω^{--} is not a rational number.

Proof: Assume the contrary. In other words, assume that there is a description which produces the digits of Ω^{--} in sequence. I will now show that if this were possible, we could solve the Halting Problem. The process for solving the Halting Problem if Ω^{--} were computable is the following:

1. Given a description, compute its position in the list of descriptions;
2. Generate the digits of Ω^{--} until you get to the digit corresponding to the given description;
3. If the digit is 1, output that the description halts. If it is 0, output that the description does not halt.

The first step in the description consists of running through all possible sequences in size-alphabetical order and keeping only those that are valid English descriptions until you finally get the description that you want. This is not a very speedy procedure, but it is specified clearly and can, in principle, be carried out.

Theorem 7 shows that the Halting Problem cannot be solved, so it would follow that Ω^{--} is not computable.

There are a variety of stronger statements that we can make about Ω^{--}. In particular, we can show that the algorithmic complexity of the first N digits of Ω^{--} is roughly $\log N$.

Theorem 10: (Complexity of Ω^{--} Theorem). If English is represented using an alphabet of $Q(>2)$ letters, then the string composed of the first Q^{N+1} digits of Ω^{--} has algorithmic complexity at least $N - C$, where C is some constant.

Proof: We use Q to denote the number of different characters in English to permit this proof to apply to other languages and to permit different people to select different character sets for English.

There is 1 0-letter English description, at most Q 1-letter English descriptions, at most Q^2 2-letter English descriptions, etc. Thus, all English descriptions of length N or less come within the first $1 + Q + Q^2 + \ldots + Q^N < Q^{N+1}$ positions on the list of descriptions. If we know the first Q^{N+1} digits of Ω^{--} we know which descriptions of length N or less halt. Now we can carry out all steps in all the English descriptions of length N or less that halt and find all strings that have complexity N or less. We simply generate a string that is not on the list of generated strings to get a string of complexity greater than N. Since describing this process takes C characters and we created a string of complexity $>N$, it follows that the complexity of the digits of Ω^{--} used here must exceed $N - C$.

PROPERTIES OF Ω

In this section I want to briefly describe some of the properties of Ω. Like Ω^{--}, Ω is uncomputable. Since Ω is a compressed version of Ω^{--}, the complexity of its finite prefixes is substantially higher. The complexity of the first N digits of Ω is essentially N.

Furthermore, Ω is truly random when represented in any base. This means that its sequence of digits will pass any statistical test that can be devised. I will not go into detail, but this implies, among other things, that every digit appears essentially the same proportion of the time as any other digit. Similarly, all possible pairs of digits, triples of digits, etc., appear with the expected frequency.

This last observation leads to the concept of a *Chaitin-random string*. A finite string is Chaitin-random when its complexity is roughly equal to its length. An infinite string is Chaitin-random if all of its finite prefixes are Chaitin-random. It can be shown that Chaitin-random infinite strings are exactly those that pass all statistical tests.

It should be noted that most computer programming languages offer pseudorandom number generators. Since these pseudorandom number generating routines are generally short, it follows that the sequences they generate are not very complex, and consequently the sequences they produce are not Chaitin-random. As a consequence, it follows that no current pseudorandom number generator will pass all statistical tests! For some modeling applications, this might have serious repercussions.

IMPLICATIONS OF AIT

There are several practical consequences of AIT that I would like to highlight briefly. First and foremost is the observation that in some sense, logic does not produce more than you put in. Results such as the argument given following Theorem 5 can be extended to show that axiom systems cannot prove consequences that are "significantly more complicated" than the axiom system.

Such results are of interest in mathematics, but they apply to "practical" axiom systems as well. In particular, legal systems and economic systems can be thought of as axiomatic systems. If one believes that there is no a priori limit on the complexity of possible behavior in a legal or economic system it would appear that no system of laws or economic doctrines are likely to capture and control the behavior of the systems.

Because of the limitations of logical systems, the exploration of which propelled Gregory Chaitin into this field, it appears that there is always the need for searching and discovery in mathematics and other domains of inquiry. Thus, the human mind will always have frontiers to explore.

ACKNOWLEDGMENTS

I would like to acknowledge many interesting conversations with Gregory Chaitin, Cristian Calude, and John Casti on the subject of AIT.

SUGGESTED READING

- J. D. Barrow: Theories of everything. Oxford University Press, 1991.
- J. D. Barrow: Pi in the sky. Oxford University Press, 1992.
- J. L. Casti: Searching for certainty. Morrow, 1990.
- J. L. Casti: Five golden rules. Wiley, New York, 1996.
- P. Davies: The mind of God. Simon & Schuster, 1992.
- G. Markowsky [8] and related papers can be found on-line at http://hyperg.iicm.tu-graz.ac.at/0x811b9908_0x0023389b;sk=DC051A05.
- D. Ruelle: Chance and chaos. Princeton University Press, Princeton, NJ, 1991.

REFERENCES

1. C. Calude: Information and randomness. Springer-Verlag, Berlin, 1994.
2. G. Chaitin: Information-theoretic incompleteness. World Scientific, Singapore, 1992.
3. G. Chaitin: Algorithmic information theory. 2nd edition, preprint, August 1993. First edition published by Cambridge University Press, 1987.
4. G. Chaitin. The limits of mathematics. Short Course given at University of Maine, June 17, 1994.
5. G. Chaitin, A. Arslanov, C. Calude: Program-size complexity computes the halting problem. EATCS Bull. 57: pp. 198-200, 1995.
6. A. I. Khinchin: Mathematical foundations of information theory. Dover Publications, New York, 1957.
7. M. Li and P. Vitányi: An introduction to Kolmogorov complexity and its applications. Springer-Verlag, New York, 1993.
8. G. Markowsky: Introduction to algorithmic information theory. J. Universal Computer Science 2(5): pp. 245-269, 1996. (Other papers on AIT are also available in this issue of JUCS.)