Algorithmic Entropy

Timothy Murphy

Course MA346H
2013

Alan Mathison Turing

Table of Contents

Introduction

In Shannon's theory, entropy is a property of a statistical ensemble—the entropy of a message depends not just on the message itself, but also on its position as an event in a probability space. Kolmogorov and Chaitin showed (independently that entropy is in fact an *intrinsic* property of an object, which can be computed (in theory at least) without reference to anything outside that object.

It is a truism of popular science that the 20th century saw 2 great scientific revolutions—Albert Einstein's General Theory of Relavity, and the Quantum Theory of Paul Dirac and others.

But as we look back at the end of the century over a longer perspective, 2 more recent innovations—Claude Shannon's Information Theory, and Alan Turing's Theory of Computability—begin to loom as large as those earlier ones.

Each of these has in its own way changed our perception of the external and internal world: information theory, with its image of information as a substance flowing through channels of communication; and Turing machines as models for algorithms and perhaps for thought itself.

Algorithmic Information Theory is the child of these 2 theories. Its basic idea is very simple. The Informational Content, or Entropy, of an object is to be measured by *the size of the smallest computer program describing that object.*

(By 'object' we mean a finite object—perhaps the state of all the particles in the universe—which can be represented as computer data, that is, as a string of bits.)

Thus an object has low entropy if it can be described succinctly. This implies that the object is highly ordered (eg "a million molecules all in state 1"). As in Statistical Mechanics, entropy can be regarded as a measure of disorder, or randomness. *High entropy corresponds to disorder; low entropy to order.*

This definition of entropy throws up a number of questions.

In the first place, what do we mean by a computer program? That is answered easily enough: we take the Turing machine as our model computer,

with its input as 'program', since Turing showed that anything any other computer can do one of his machines could do as well.

But even so, there still seems to be a problem: given any finite object, however complicated, we could always construct a 'one-off' Turing machine which would output a description of the object without any input at all.

Turing comes to our rescue again. He showed that a single *universal* machine can emulate every other machine. We may therefore insist that this universal machine should be used in computing the entropy of an object.

In Chapters 1–3 we recall the main features of Turing's theory, which we then apply in Chapter 4 to give a precise definition of Algorithmic Entropy.

Shannon showed that a message M can be compressed until it consists of *pure information*, as measured by its entropy $H(M)$. Algorithmic Information Theory turns this idea on its head—the compressibility of an object is taken as a measure of its entropy.

Chapter 1

The Anatomy of a Turing Machine

We follow Chaitin in adopting a slight variation on Turing's original machine.

Turing's model of an ideal computer was remarkably concise, largely because he used a single tape for input, output and 'scratch' calculations.

However, Turing's work showed—paradoxically perhaps—that almost any model with unlimited memory would serve as well as his own; at least, so long as *computability* was the only question at issue. Once one steps beyond that—which we shall not do—into the realms of complexity or efficiency, the equivalence of different models is by no means so clear.

For our purpose, Turing's model is *too* concise, since we need to distinguish more clearly between input and output.

The Chaitin model that we shall follow shares the following essential features of Turing's machine:

1. The machine 'moves' in discrete steps $0, 1, 2, \ldots$, which we may call *moments*.

2. The 'working part' of the machine is a doubly infinite tape, divided into squares numbered by the integers: Each square contains one of the bits $0, 1$. At any moment all but a finite number of the bits are 0.

   Mathematically, the tape contents are defined by a map

   $$t : \mathbb{Z} \to \{0, 1\},$$

   where

   $$t(i) = 0 \text{ if } |i| > N$$

   for some N.

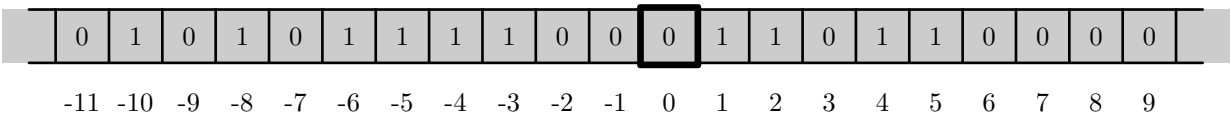

Figure 1.1: Tape with numbered squares

3. At each moment the machine is in one of a finite number of *states*

$$q_0, q_1, \ldots, q_n.$$

State q_0 is both the initial and the final, or halting, state. The computation is complete if and when the machine re-enters this state.

We denote the set of states by

$$Q = \{q_0, q_1, \ldots, q_n\}.$$

4. Both the *action* that the machine takes at each moment, and the *new state* that it moves into, depend entirely on 2 factors:

   (a) the state q of the machine;

   (b) the bit $t(0)$ on square 0 of the tape.

   It follows that a machine T can be described by a single map

$$T : Q \times \{0, 1\} \to A \times Q,$$

   where A denotes the set of possible actions.

   To emphasize that at any moment the machine can only take account of the content of square 0, we imagine a 'scanner' sitting on the tape, through which we see this square.

Turning to the *differences* between the Chaitin and Turing models:

1. Turing assumes that the input x to the machine is written directly onto the tape; and whatever is written on the tape if and when the machine halts is the output $T(x)$.

   In the Chaitin model, by contrast, the machine reads its input from an *input string*, and writes its output to an *output string*.

2. Correspondingly, the possible actions differ in the 2 models. In the Turing model there are just 4 actions:

$$A = \{\texttt{noop}, \texttt{swap}, \longleftarrow, \longrightarrow\}.$$

Each of these corresponds to a map $\mathcal{T} \to \mathcal{T}$ from the set $\mathcal{T}$ of all tapes (ie all maps $\mathbb{Z} \to \{0,1\}$) to itself. Thus `noop` leaves the content of square 0 unchanged (though of course the state may change at the same moment), `swap` changes the value of $t(0)$ from 0 to 1, or 1 to 0, while $\longleftarrow$ and $\longrightarrow$ correspond to the mutually inverse shifts

$$(\longleftarrow t)(i) = t(i+1), \quad (\longrightarrow t)(i) = t(i-1),$$

Informally, we can think of $\longleftarrow$ and $\longrightarrow$ as causing the scanner to move to the left or right along the tape.

In the Chaitin model we have to add 2 more possible actions:

$$A = \{\texttt{noop}, \texttt{swap}, \longleftarrow, \longrightarrow, \texttt{read}, \texttt{write}\}.$$

The action `read` takes the next bit b of the input string and sets $t(0) = b$, while `write` appends $t(0)$ to the output string. (Note that once the bit b has been read, it cannot be re-read; the bit is removed from the input string. Similarly, once `write` has appended a bit to the output string, this cannot be deleted or modified. We might say that the input string is a 'read-only stack', while the output string is a 'write-only stack'.)

To summarise, while the Turing model has in a sense no input/output, the Chaitin model has input and output 'ports' through which it communicates with the outside world.

Why do we need this added complication? We are interested in the Turing machine as a tool for converting an input string into an output string. Unfortunately it is not clear how we translate the content of the tape into a string, or vice versa. We could specify that the string begins in square 0, but where does it end? How do we distiguish between 11 and 110?

We want to be able to measure the *length* of the input program or string p, since we have defined the entropy of s to be the length of the shortest p for which $T(p) = s$. While we could indeed define the length of an input tape in the Turing model to be (say) the distance between the first and last 1, this seems somewhat artificial. It would also mean that the input and output strings would have to end in 1's; outputs 111 and 1110 would presumably be indistinguishable.

Having said all that, one must regret to some extent departing from the standard; and it could be that the extra work involved in basing Algorithmic Information Theory on the Turing model would be a small price to pay.

However that may be, for the future we shall use the Chaitin model exclusively; and henceforth we shall always use the term "Turing machine" to mean *the Chaitin model of the Turing machine.*

1.1 Formality

Definition 1.1. *We set*

$$\mathbb{B} = \{0, 1\}.$$

Thus $\mathbb{B}$ is the set of possible *bits*.

We may sometimes identify $\mathbb{B}$ with the 2-element field $\mathcal{F}_2$, but we leave that for the future.

Definition 1.2. *A Turing machine T is defined by giving a finite set Q, and a map*

$$T : Q \times \mathbb{B} \to A \times Q,$$

where

$$A = \{\texttt{noop}, \texttt{swap}, \longleftarrow, \longrightarrow, \texttt{read}, \texttt{write}\}.$$

1.2 The Turing machine as map

We denote the set of all finite strings by $\mathbb{S}$.

Definition 1.3. *Suppose T is a Turing machine. Then we write*

$$T(p) = s,$$

where $p, s \in \mathbb{S}$, if the machine, when presented with the input string p, reads it in completely and halts, having written out the string s. If for a given p there is no s such that $T(p) = s$ then we say that $T(p)$ is undefined, and write

$$T(p) = \perp,$$

Remark. Note that T must read to the *end* of the string p, and then halt. Thus $T(p) = \perp$ (ie $T(p)$ is undefined) if the machine halts before reaching the end of p, if it tries to read beyond the end of p, or if it never halts, eg because it gets into a loop.

Example. As a very simple example, consider the 1-state machine T (we will generally ignore the starting state q_0 when counting the number of states) defined by the following rules:

$$\begin{aligned} (q_0, 0) &\mapsto (\texttt{read}, q_1) \\ (q_1, 0) &\mapsto (\texttt{write}, q_0) \\ (q_1, 1) &\mapsto (\texttt{read}, q_1) \end{aligned}$$

This machine reads from the input string until it reads a 0, at which point it outputs a 0 and halts. (Recall that q_0 is both the initial and the final, or halting, state; the machine halts if and when it re-enters state q_0.)

Thus $T(p)$ is defined, with value 0, if and only if p is of the form

$$p = 1^n 0.$$

Remark. By convention, if no rule is given for (q_i, b) then we take the rule to be $(q_i, b) \to (\texttt{noop}, 0)$ (so that the machine halts).

In the above example, no rule is given for $(q_0, 1)$. However, this case can never arise; for the tape is blank when the machine starts, and if it re-enters state q_0 then it halts.

It is convenient to extend the set $\mathbb{S}$ of strings by setting

$$\mathbb{S}^\perp = S \cup \{\perp\}.$$

By convention we set

$$T(\perp) = \perp$$

for any Turing machine T. As one might say, "garbage in, garbage out".

With this convention, every Turing machine defines a map

$$T : \mathbb{S}^\perp \to \mathbb{S}^\perp.$$

1.3 The Church-Turing Thesis

Turing asserted that *any effective calculation can be implemented by a Turing machine.* (Actually, Turing spoke of the Logical Computing Machine, or LCM, the term 'Turing machine' not yet having been introduced.)

About the same time — in 1936 — Church put forward a similar thesis, couched in the rather esoteric language of the *lambda calculus.* The two ideas were rapidly shown to be equivalent, and the term 'Church-Turing thesis' was coined, although from our point of view it would be more accurate to call it the Turing thesis.

The thesis is a philosophical, rather than a mathematical, proposition, since the term 'effective' is not — perhaps cannot be — defined. Turing explained on many occasions what he meant, generally taking humans as his model, eg "A man provided with paper, pencil, and rubber, and subject to strict discipline, is in effect a universal machine." or "The idea behind digital computers may be explained by saying that these machines are intended to carry out any operations which could be done by a human computer."

The relevance of the Church-Turing thesis to us is that it gives us confidence—if we believe it—that any algorithmic procedure, eg the Euclidean algorithm for computing $\gcd(m, n)$, can be implemented by a Turing machine.

It also implies that a Turing machine cannot be 'improved' in any simple way, eg by using 2 tapes or even 100 tapes. Any function $T : \mathbb{S}^{\perp} \to \mathbb{S}^{\perp}$ that can be implemented with such a machine could equally well be implemented by a standard 1-tape Turing machine. What we shall later call the class of computable or Turing functions appears to have a natural boundary that is not easily breached.

Chapter 2

Prefix-free codes

2.1 Domain of definition

Definition 2.1. *The* domain of definition *of the Turing machine T is the set*

$$\Omega(T) = \{p \in \mathbb{S} : T(p) \neq \bot\}.$$

In other words, $\Omega(T)$ is the set of strings p for which $T(p)$ is defined.

Recall that $T(p)$ may be undefined in three different ways:

Incompletion The computation may never end;

Under-run The machine may halt before reading the entire input string p;

Over-run There may be an attempt to read beyond the end of p.

We do not distinguish between these 3 modes of failure, writing

$$T(p) = \bot.$$

in all three cases.

As we shall see, this means that $T(p)$ can only be defined for a restricted range of input strings p. At first sight, this seems a serious disadvantage; one might suspect that Chaitin's modification of the Turing machine had affected its functionality. However, as we shall see, we can avoid the problem entirely by encoding the input; and this even turns out to have incidental advantages, for example extending the theory to objects other than strings.

2.2 Prefix-free sets

Definition 2.2. *Suppose $s, s' \in \mathbb{S}$. We say that s is a* prefix *of s', and we write $s \prec s'$, if s is an initial segment of s', ie if*

$$s' = b_0 b_1 \dots b_n$$

then

$$s = b_0 b_1 \dots b_r$$

for some $r \leq n$.

Evidently

$$s \prec s' \Longrightarrow |s| \leq |s'| \,.$$

Definition 2.3. *A subset $S \subset \mathbb{S}$ is said to be* prefix-free *if*

$$s \prec s' \Longrightarrow s = s'$$

for all $s, s' \in S$.

In other words, S is prefix-free if no string in S is a prefix of another string in S.

Theorem 2.1. *The domain of definition of a Turing machine T,*

$$\Omega(T) = \{s \in \mathbb{S} : T(s) \neq \perp\},$$

is prefix-free.

Proof ▶. Suppose $s' \prec s,\ s' \neq s$. Then if $T(s')$ is defined, the machine must halt after reading in s, and so it cannot read in the whole of s. Hence $T(s)$ is undefined.

Proposition 2.1. *If $S \subset \mathbb{S}$ is prefix-free then so is every subset $T \subset S$.*

Proposition 2.2. *A prefix-free subset $S \subset \mathbb{S}$ is* maximal *(among prefix-free subsets of $\mathbb{S}$) if and only if each $t \in \mathbb{S}$ is either a prefix of some $s \in S$ or else some $s \in S$ is a prefix of t.*

Remark. For those of a logical turn of mind, we may observe that being prefix-free is a *property of finite character*, that is, a set S is prefix-free if and only if that is true of every finite subset $F \subset S$. It follows by Zorn's Lemma that each prefix-free set S is contained in a maximal prefix-free set. However, we shall make no use of this fact.

2.3 Prefix-free codes

Definition 2.4. *A* coding *of a set X is an injective map*

$$\gamma : X \to \mathbb{S}.$$

The coding γ is said to be prefix-free *if its image*

$$\operatorname{im}\gamma \subset \mathbb{S}$$

is prefix-free.

By encoding X, we can in effect take the elements $x \in X$ as input for the computation $T(\gamma x)$; and by choosing a prefix-free encoding we allow the possibility that the computation may complete for all $x \in X$.

2.4 Standard encodings

It is convenient to adopt *standard* prefix-free encodings for some of the sets we encounter most often, for example the set $\mathbb{N}$ of natural numbers, or the set of Turing machines. In general, whenever we use the notation $\langle x \rangle$ without further explanation it refers to the standard encoding for the set in question.

2.4.1 Strings

Definition 2.5. *We encode the string*

$$s = b_1 b_2 \cdots b_n \in \mathbb{S}.$$

as

$$\langle s \rangle = 1 b_1 1 b_2 1 \cdots 1 b_n 0.$$

Thus a 1 in odd position signals that there is a string-bit to follow, while a 0 in odd position signals the end of the string.

Example. If $s = 01011$ then

$$\langle s \rangle = 10111011110.$$

If $s = \square$ (the empty string) then

$$\langle s \rangle = 0.$$

Definition 2.6. *We denote the length of the string $s \in \mathbb{S}$, ie the number of bits in s, by $|s|$.*

Evidently

$$|s| = n \Longrightarrow |\langle s \rangle| = 2n + 1.$$

Proposition 2.3. *The map*

$$s \mapsto \langle s \rangle : \mathbb{S} \to \mathbb{S}$$

defines a maximal prefix-free code for $\mathbb{S}$.

Proof ▶. A string is of the form $\langle s \rangle$ if and only if

1. it is of odd length,
2. the last bit is 0, and
3. this is the only 0 in an odd position.

The fact that $\langle s \rangle$ contains just one 0 in odd position, and that at the end, shows that the encoding is prefix-free.

To see that it is *maximal*, suppose $x \in \mathbb{S}$ is not of the form $\langle s \rangle$ for any $s \in \mathbb{S}$. We need only look at the odd bits of x. If there is no 0 in odd position then appending 0 or 00 to x (according as x is of even or odd length) will give a string of form $\langle s \rangle$. If there is a 0 in odd position, consider the first such. If it occurs at the end of x then x is of form $\langle s \rangle$, while if it does not occur at the end of x then the prefix up to this 0 is of the form $\langle s \rangle$ for some s.

It follows that if x is not already of the form $\langle s \rangle$ then it cannot be appended to the set $\{\langle s \rangle : s \in \mathbb{S}\}$ without destroying the prefix-free property of this set.

2.4.2 Natural numbers

Definition 2.7. *Suppose* $n \in \mathbb{N}$. *Then we define* $\langle n \rangle$ *to be the string*

$$\langle n \rangle = \overbrace{1 \cdots 1}^{n \text{ 1's}} 0.$$

Example.

$$\langle 3 \rangle = 1110$$
$$\langle 0 \rangle = 0.$$

Proposition 2.4. *The map*

$$n \mapsto \langle n \rangle : \mathbb{N} \to \mathbb{S}$$

defines a maximal prefix-free code for $\mathbb{N}$.

2.4.3 Turing machines

Recall that a Turing machine T is defined by a set of rules

$$R : (q, b) \mapsto (a, q').$$

We encode this rule in the string

$$\langle R \rangle = \langle q \rangle b \langle a \rangle \langle q' \rangle,$$

where the 6 actions are coded by 3 bits as follows:

$$\begin{array}{ll} \texttt{noop} & \mapsto 000 \\ \texttt{swap} & \mapsto 001 \\ \longleftarrow & \mapsto 010 \\ \longrightarrow & \mapsto 011 \\ \texttt{read} & \mapsto 100 \\ \texttt{write} & \mapsto 101 \end{array}$$

So for example, the rule $(1, 1) \mapsto (\longleftarrow, 2)$ is coded as

$$1011010110.$$

Definition 2.8. *Suppose the Turing machine T is specified by the rules $R_1, \ldots, R_n$. Then we set*

$$\langle T \rangle = \langle n \rangle \langle R_1 \rangle \cdots \langle R_n \rangle.$$

We do not insist that all the rules are given, adopting the convention that if no rule is given for (q, b) then the 'default rule'

$$(q, b) \mapsto (\texttt{noop}, 0)$$

applies.

Also, we do not specify the order of the rules; so different codes may define the same machine.

Instead of prefacing the rules with the number $\langle n \rangle$ of rules we could equally well signal the end of the rules by giving a rule with non-existent action, eg 111. Thus

$$001110$$

could serve as a stop-sign.

2.4.4 Product sets

Proposition 2.5. *If S and S' are both prefix-free subsets of $\mathbb{S}$ then so is*

$$SS' = \{ss' : s \in S, s' \in S'\},$$

where ss' denotes the concatenation of s and s'.

Proof ▶. If $s_1 s_1' \prec s_2 s_2'$ then either (a) $s_1 \prec s_1'$, or (b) $s_1' \prec s_1$, or (c) $s_1 = s_1'$ and either $s_2 \prec s_2'$ or $s_2' \prec s_2$.

This gives a simple way of extending prefix-free codes to product-sets. For example, the set $\mathbb{S}^2 = \mathbb{S} \times \mathbb{S}$ of pairs of strings can be coded by

$$(s_1, s_2) \mapsto \langle s_1 \rangle \langle s_2 \rangle.$$

Or again—an instance we shall apply later—the set $\mathbb{S} \times \mathbb{N}$ can be coded by

$$(s, n) \mapsto \langle s \rangle \langle n \rangle.$$

2.4.5 A second code for $\mathbb{N}$

Definition 2.9. *Suppose $n \in \mathbb{N}$. Then we set*

$$[n] = \langle B(n) \rangle,$$

where $B(n)$ denotes the binary code for n.

Example. Take $n = 6$. Then

$$B(n) = 110,$$

and so

$$[6] = 1111100.$$

Proposition 2.6. *The coding $[n]$ is a maximal prefix-free code for $\mathbb{N}$.*

The conversion from one code for $\mathbb{N}$ to the other is clearly 'algorithmic'. So according to the Church-Turing thesis, there should exist Turing machines S, T that will convert each code into the other:

$$S([n]) = \langle n \rangle, \quad T(\langle n \rangle) = [n].$$

We construct such a machine T in Appendix **??**. (We leave the construction of S to the reader ....) As we shall see, it may be obvious but it is not simple!

Summary

We have adopted Chaitin's model of a Turing machine. The set $\Omega(T)$ of input strings, or *programs*, for which such a machine T is defined constitutes a *prefix-free* subset of the set $\mathbb{S}$ of all strings.

Chapter 3

Universal Machines

CERTAIN TURING MACHINES have the remarkable property that they can emulate, or imitate, all others.

3.1 Universal Turing machines

We have defined a code $\langle T \rangle$ for Turing machines in the last Chapter (Subsection 2.4.3.

Definition 3.1. *We say that the Turing machine U is* universal *if*

$$U(\langle T \rangle p) = T(p)$$

for every Turing machine T and every string $p \in \mathbb{S}$.

Informally, a universal Turing machine can 'emulate' any Turing machine.

There is another definition of a universal machine which is sometimes seen: the machine U is said to be universal if given any Turing machine T we can find a string $s = s(T)$ such that

$$U(sp) = T(p)$$

for all $p \in \mathbb{S}$. Evidently a universal machine according to our definition is universal in this sense; and since we are only concerned with the *existence* of a universal machine it does not matter for our purposes which definition is taken.

Theorem 3.1. *There exists a universal Turing machine.*

We shall outline the construction of a universal machine in Chapter **??**. The construction—which is rather complicated—can be divided into 4 parts:

1. We instroduct a variant of Turing machines, using stacks instead of a tape.

2. We show that '2 stacks suffice', with these stacks replacing the left and right halves of the tape.

3. We show that $n \geq 2$ stacks is equivalent to 2 stacks, in the sense that given an n-stack machine S_n we can always find a 2-stack machine S_2 such that
$$S_2(p) = S_n(p)$$
for all inputs p.

4. We show that a universal machine can be implemented as a 4-stack machine. 2 of the stacks being used to store the rules of the machine being emulated.

But for the moment we merely point out that the Church-Turing thesis suggests that such a machine must exist; for it is evident that given the rules defining a machine T, and an input string p, the 'human computer' can determine the state of the machine and the content of the tape at each moment $t = 0, 1, 2, \ldots$, applying the appropriate rule to determine the state and tape-content at the subsequent moment.

There are many universal machines. Indeed, a universal machine U can emulate itself. So the machine V defined by
$$V(p) = U(\langle U \rangle p)$$
is also universal.

It is an interesting question to ask if there is a *best* universal machine in some sense of 'best'. One might ask for the universal machine with the minimal number of states; for there are only a finite number of Turing machines with $\leq n$ states, since there are only a finite number of rules
$$(q_i, b) \mapsto (a, q_o)$$
with $0 \leq q_i, q_o \leq n$. This question has some relevance for us, since we shall define algorithmic entropy (shortly) *with respect to a given universal machine*. It will follow from the fact that two universal machines U, V can emulate each other that the entropies with respect to U and V cannot differ by more than a constant $C = C(U, V)$. But it would be nice to have a concept of *absolute* algorithmic entropy.

Summary

The universal machine U performs exactly like the machine T, provided the program is prefixed by the code $\langle T \rangle$ for the machine:

$$U\left(\langle T \rangle p\right) = T(p).$$

Chapter 4

Algorithmic Entropy

WE ARE NOW in a position to give a precise definition of the *Algorithmic Entropy* (or *Informational Content*) $H(s)$ of a string s.

4.1 The entropy of a string

Definition 4.1. *The algorithmic entropy $H_T(s)$ of a string s with respect to the Turing machine T is defined to be the length of the shortest input p which, when fed into the machine T, produces the output s*

$$H_T(s) = \min_{p:T(p)=s} |p|.$$

If there is no such p then we set $H_T(s) = \infty$.

Now let us choose, once and for all, a particular universal Turing machine U as our 'model'. The actual choice, as we shall see, is irrelevant *up to a constant*.

Definition 4.2. *The algorithmic entropy $H(s)$ of the string s is defined to be the entropy with respect to our chosen universal machine U*

$$H(s) = H_U(s).$$

Informational content and *Kolmogorov complexity* are alternative terms for 'algorithmic entropy', which we shall often abbreviate to *entropy*.

Proposition 4.1. *For each string s,*

$$0 < H(s) < \infty.$$

Proof ▶. The universal machine U cannot halt before it has read any input; for in that case its output would always be the same, and it could not possibly emulate every machine T. Thus $H(s) > 0$.

On the other hand, given a string s we can certainly construct a machine T which will output s, say without any input. But then U outputs s on input $p = \langle T \rangle$. Hence $H(s) < \infty$.

It is often convenient to choose a particular minimal input p for a given string s with respect to a given machine T.

Definition 4.3. *Suppose $H_T(s) < \infty$. Then we denote by $\mu_T(s)$ that input p of minimal length for which $T(p) = s$, choosing the first such p in lexicographical order in case of ambiguity; and we set*

$$\mu(s) = \mu_U(s).$$

We shall sometimes call $\mu_T(s)$ or $\mu(s)$ the *minimal program* for s.

Proposition 4.2. *For each machine T,*

$$H(s) \leq H_T(s) + O(1),$$

ie $H(s) \leq H_T(s) + C$, where $C = C(T)$ is independent of s.

Proof ▶. Let $\pi = \mu_T(s)$, so that

$$T(\pi) = s, \quad |\pi| = H_T(s).$$

Then

$$U(\langle T \rangle \pi) = T(\pi) = s,$$

where $\langle T \rangle$ is the code for T. Hence

$$\begin{aligned} H(s) &\leq |\langle T \rangle| + |\pi| \\ &= |\langle T \rangle| + H(s) \\ &= H(s) + C. \end{aligned}$$

Proposition 4.3. *If V is a second universal machine, then*

$$H_V(s) = H(s) + O(1).$$

Proof ▶. By the last Proposition

$$H(s) \leq H_V(s) + O(1).$$

But by the same argument, taking V as our chosen universal machine,

$$H_V(s) \leq H(s) + O(1).$$

4.2 Entropy of a number

The concept of algorithmic entropy extends to any object that can be encoded as a string.

Definition 4.4. *Suppose $\gamma : X \to \mathbb{S}$ is a coding for objects in the set X. Then we define the entropy of $x \in X$ with respect to this encoding as*

$$H^\gamma(x) = H(\gamma(x)).$$

Where we have defined a standard encoding for a set, we naturally use that to define the entropy of an object in the set.

Definition 4.5. *The algorithmic entropy of the natural number $n \in \mathbb{N}$ is*

$$H(n) = H(\langle n \rangle).$$

It is worth noting that the entropy of a string is unchanged (up to a constant) by encoding.

Proposition 4.4. *We have*

$$H(\langle s \rangle) = H(s) + O(1).$$

Proof ▶. It is easy to modify U (we need only consider `write` statements) to construct a machine X which outputs $\langle s \rangle$ when U outputs s, ie

$$U(\mu) = s \Longrightarrow X(\mu) = \langle s \rangle.$$

If now $\mu = \mu(s)$ then

$$\begin{aligned} H(\langle s \rangle) &\leq |\mu| + |\langle X \rangle| \\ &= H(s) + |\langle X \rangle| \\ &= H(s) + O(1). \end{aligned}$$

Conversely, we can equally easily modify U to give a machine Y which outputs s when U outputs $\langle s \rangle$; and then by the same argument

$$H(s) \leq H(\langle s \rangle) + O(1).$$

4.3 Equivalent codes

As we have said, given a coding of a set X we can define the algorithmic entropy $H(x)$ of the elements $x \in X$, But what if we choose a different encoding?

The Church-Turing thesis (Section 1.3) allows us to hope that $H(x)$ will be independent of the coding chosen for X — provided we stick to 'effective' codes.

The following definition should clarify this idea.

Definition 4.6. *The two encodings*

$$\alpha, \beta : X \to \mathbb{S}$$

are said to be Turing-equivalent *if there exist Turing machines S, T such that*

$$S(\langle \alpha x \rangle)) = \beta x, \quad T(\langle \beta x \rangle)) = \alpha x$$

for all $x \in X$.

Proposition 4.5. *If α, β are Turing-equivalent encodings for X then*

$$H(\alpha x) = H(\beta x) + O(1).$$

Proof ▶. Suppose

$$S(\langle \alpha x \rangle) = \beta x, \quad T(\langle \beta x \rangle) = \alpha x.$$

Let

$$\mu = \mu(\alpha x), \quad \nu = \mu(\beta x),$$

so that

$$U(\mu) = \alpha x, \quad U(\nu) = \beta x.$$

We can construct a machine X which starts by emulating U, except that it saves the output s in coded form $\langle s \rangle$. Then when U has completed its computation, X emulates the machine S, taking $\langle s \rangle$ as input. Thus

$$X(s') = S(\langle U(s') \rangle).$$

In particular,

$$X(\mu) = \beta x,$$

and so

$$\begin{aligned} H(\beta x) &\leq |\mu| + |\langle X \rangle| \\ &= H(\alpha x) + O(1); \end{aligned}$$

and similarly in the other direction.

4.3.1 The binary code for numbers

As an illustration of these ideas, recall that in Section 2.4.5) we introduced a second 'binary' encoding

$$[n] = \langle B(n) \rangle$$

for the natural numbers.

Proposition 4.6. *The binary code $[n]$ for the natural numbers is Turing-equivalent to the standard code $\langle n \rangle$.*

Proof ▶. We have to construct Turing machines S, T such that

$$S([n]) = \langle n \rangle, \quad T(\langle \langle n \rangle \rangle) = B(n).$$

The construction of a suitable machine T is detailed in Appendix **??**. The (simpler) construction of S is left as an exercise to the reader.

Corollary 4.1. *The algorithmic entropies with respect to the two encodings of $\mathbb{N}$ are the same, up to a constant:*

$$H([n]) = H(n) + O(1).$$

Thus it is open to us to use the binary encoding or the standard encoding, as we prefer, when computing $H(n)$.

4.4 Joint entropy

We end this chapter by considering some basic properties of algorithmic entropy.

Definition 4.7. *The joint entropy $H(s,t)$ of 2 strings s and t is defined to be*

$$H(s,t) = H\left(\langle s \rangle \langle t \rangle\right).$$

Note that $H(s,t)$ measures all the information in s and t; for we can recover s and t from $\langle s \rangle \langle t \rangle$. We could not set $H(s,t) = H(st)$, ie we cannot simply concatenate s and t since this would lose the information of where the split occurred.

Example. Suppose

$$s = 1011, \ t = 010.$$

Then $H(s,t) = H(s * t)$, where

$$s * t = 11101111001011100000.$$

Proposition 4.7. *Joint entropy is independent of order:*

$$H(t,s) = H(s,t) + O(1).$$

Proof ▶. We can construct a Turing machine T which converts $\langle t\rangle\langle s\rangle$ into $\langle s\rangle\langle t\rangle$ for any strings s and t.

Now suppose p is a minimal program for $\langle s\rangle\langle t\rangle$: $U(p) = \langle s\rangle\langle t\rangle$. Let the machine M implement T followed by U:

$$M = U \circ T.$$

Then $M(p) = \langle t\rangle\langle s\rangle$ and so

$$\begin{aligned} H\left(\langle t\rangle\langle s\rangle\right) &\leq H_M\left(\langle t\rangle\langle s\rangle\right) + O(1) \\ &\leq |p| + O(1) \\ &= H\left(\langle s\rangle\langle t\rangle\right) + O(1). \end{aligned}$$

4.5 Conditional entropy

Informally, the conditional entropy $H(s \mid t)$ measures the additional information contained in the string s if we already 'know' the string t.

But what do we mean by 'knowing' t?

In the context of algorithmic information theory it is natural to interpret this to mean that *we are given the minimal program*

$$\tau = \mu(t) = \mu_U(t)$$

for t.

We would like to define $H(s \mid t)$ as the length of the shortest string q which when appended to τ gives us a program for s:

$$U(\tau q) = s.$$

Unfortunately, there is a flaw in this idea. *If* $U(\tau)$ *is defined then* $U(\tau q)$ *cannot be*—for U will already have halted on reading the string τ.

To circumvent this obstacle, let us recall that

$$H(s) \leq H_T(s) + |\langle T\rangle|.$$

Thus if we set

$$H'(s) = \min_T \left(H_T(s) + |\langle T\rangle|\right),$$

then on the one hand

$$H(s) \leq H'(s)$$

while on the other hand

$$\begin{aligned} H'(s) &\leq H_U(s) + |\langle U \rangle| \\ &= H(s) + |\langle U \rangle|. \end{aligned}$$

Putting these together,

$$H'(s) = H(s) + O(1).$$

So $H'(s)$ would serve in place of $H(s)$ as a measure of absolute entropy.

This trick suggests the following definition of conditional entropy.

Definition 4.8. *Suppose $s, t \in \mathbb{S}$. Let $\tau = \mu(t)$ be the minimal program for t (with respect to the universal machine U):*

$$U(\tau) = t, \quad |\tau| = H(t).$$

Then we define the conditional entropy *of s given t to be*

$$H(s \mid t) = \min_{T,q:T(\tau q)=s} \left(|q| + |\langle T \rangle| \right).$$

Proposition 4.8. *Conditional entropy does not exceed absolute entropy:*

$$H(s \mid t) \leq H(s) + O(1).$$

Proof ▶. Let π be the minimal program for s:

$$U(\pi) = s, \quad |\pi| = H(s).$$

We can construct a machine T such that

$$T(pq) = U(q)$$

for any $p \in \Omega(U)$.

The machine T starts by imitating U 'silently' and 'cleanly', ie without output, and in such a way that the internal tape is cleared after use. Then when U would halt, T imitates U again, but this time in earnest.

In particular,

$$T(\tau\pi) = U(\pi) = s,$$

and so

$$\begin{aligned} H(s \mid t) &\leq |\pi| + |\langle T \rangle| \\ &= H(s) + |\langle T \rangle| \\ &= H(s) + O(1). \end{aligned}$$

Corollary 4.1. $H(s \mid t) < \infty$.

Proposition 4.9.

$$H(s \mid s) = O(1).$$

Proof ▶. In this case we *can* use the universal machine U to output s, taking $q = \Lambda$, the empty string:

$$U(\tau\Lambda) = U(\tau) = s,$$

and so

$$\begin{aligned} H(s \mid s) &\leq 0 + |\langle U \rangle| \\ &= O(1). \end{aligned}$$

Summary

We have defined the entropy $H(s)$ of a string s, as well as the conditional entropy $H(s \mid t)$ and the joint entropy $H(s,t)$ of two strings s and t. It remains to justify this terminology, by showing in particular that

$$H(s,t) = H(s) + H(t \mid s) + O(1).$$

Chapter 5

The Halting Problem

in general it is impossible to determine for which input strings p a Turing machine T will complete its computation and halt. The proof of this is reminiscent of the 'diagonal argument' used in the proof of Cantor's Theorem on cardinal numbers, which states that the number of elements of a set X is strictly less than the number of subsets of X.

5.1 Sometimes it *is* possible

It is easy to say when *some* Turing machines will halt. Consider for example our adding machine

$$T : \langle m \rangle \langle n \rangle \to \langle m + n \rangle.$$

We know that, for any Turing machine T, the input strings p for which $T(p)$ is defined form a prefix-free set.

But it is easy to see that the set

$$S = \{p = \langle m \rangle \langle n \rangle : m, n \in \mathbb{N}\}$$

is not only prefix-free, but is actually a maximal prefix-free set, that is, if any further string is added to S it will cease to be prefix-free. It follows that for our adding machine, $T(p)$ is defined precisely when $p \in S$.

Moreover, it is easy to construct a Turing machine H which 'recognises' S, ie such that

$$H(p) = \begin{cases} 1 \text{ if } p \in S, \\ 0 \text{ if } p \notin S. \end{cases}$$

This argument applies to any Turing machine

$$T : \langle n \rangle \to \langle f(n) \rangle,$$

where $f : \mathbb{N} \to \mathbb{N}$ is a computable function.

Perhaps surprisingly, there *are* Turing machines to which it does not apply.

5.2 The Halting Theorem

Proposition 5.1. *There does not exist a Turing machine H such that*

$$H(\langle T\rangle\langle p\rangle) = \begin{cases} 1 \text{ if } T(p) \text{ is defined} \\ 0 \text{ if } T(p) \text{ is undefined} \end{cases}$$

for every Turing machine T and every string p

Proof ▶. Suppose such a Turing machine H exists. (We might call it a 'universal halting machine'.)

Let us set $p = \langle T\rangle$. (In other words, we are going to feed T with its own code $\langle T\rangle$.) Then

$$H(\langle T\rangle\langle\langle T\rangle\rangle) = \begin{cases} 1 \text{ if } T(\langle T\rangle) \text{ is defined} \\ 0 \text{ if } T(\langle T\rangle) \text{ is undefined.} \end{cases}$$

Now let us modify H to construct a 'doubling machine' D such that

$$D(\langle T\rangle) = H(\langle T\rangle\langle\langle T\rangle\rangle)$$

for any machine T.

Thus D reads the input, expecting the code $\langle T\rangle$ for a Turing machine. It doesn't really matter what D does if the input is not of this form; we may suppose that it either tries to read past the end of the input, or else goes into an infinite loop. But if the input is of the form $\langle T\rangle$ then D doubles it, writing first $\langle T\rangle$ on the tape, followed by the string code $\langle\langle T\rangle\rangle$. Then it emulates H taking this as input.

Thus

$$D(\langle T\rangle) = H(\langle T\rangle\langle\langle T\rangle\rangle) = \begin{cases} 1 & \text{if } T(\langle T\rangle) \text{ is defined,} \\ 0 & \text{if } T(\langle T\rangle) \text{ is undefined.} \end{cases}$$

Finally, we modify D (at the output stage) to construct a machine X which outputs 0 if D outputs 0, but which goes into an infinite loop if D outputs 1. Thus

$$X(s) = \begin{cases} \perp & \text{if } D(s) = 1, \\ 0 & \text{if } D(s) = 0. \end{cases}$$

(We don't care what X does if $D(s)$ is not 0 or 1.) Then

$$X(\langle T\rangle) = \begin{cases} \perp & \text{if } T(\langle T\rangle) \neq \perp, \\ 0 & \text{if } T(\langle T\rangle) = \perp. \end{cases}$$

This holds for all Turing machines T, In particular it holds for the machine X itself:

$$X(\langle X\rangle) = \begin{cases} \perp & \text{if } X(\langle X\rangle) \neq \perp, \\ 0 & \text{if } X(\langle X\rangle) = \perp. \end{cases}$$

This leads to a contradiction, whether $X(\langle X\rangle)$ is defined or not. We conclude that there cannot exist a 'halting machine' H of this kind.

We improve the result above by showing that in general there does not exist a halting machine for a single Turing machine.

Theorem 5.1. *Suppose U is a universal machine. Then there does not exist a Turing machine H such that*

$$H(\langle p\rangle) = \begin{cases} 1 \textit{ if } U(p) \textit{ is defined} \\ 0 \textit{ if } U(p) \textit{ is undefined.} \end{cases}$$

Proof ▶. Substituting $\langle X\rangle p$ for p, we have

$$H(\langle\langle X\rangle p\rangle) = \begin{cases} 1 \text{ if } X(p) \text{ is defined} \\ 0 \text{ if } X(p) \text{ is undefined.} \end{cases}$$

Evidently we can construct a slight variant H' of H which starts by decoding $\langle\langle X\rangle p\rangle$, replacing it by $\langle X\rangle\langle p\rangle$, and then acts like H, so that

$$H'(\langle X\rangle\langle p\rangle) = \begin{cases} 1 \text{ if } X(p) \text{ is defined} \\ 0 \text{ if } X(p) \text{ is undefined.} \end{cases}$$

But we saw in the Proposition above that such a machine cannot exist.

Chapter 6

Recursive sets

A set of string is recursive if it can be 'recognised' by a computer. Recursive sets offer an alternative approach to computability. The concept of recursive enumerablity is more subtle, and links up with the Halting Problem.

6.1 Recursive sets

Definition 6.1. *A set of strings $S \subset \mathbb{S}$ is said to be* recursive *if there exists a Turing machine A such that*

$$A(\langle s \rangle) = \begin{cases} 1 & \text{if } s \in S \\ 0 & \text{if } s \notin S. \end{cases}$$

We say that A is an *acceptor* for S, or that A *recognises* S.

Note that if A is an acceptor then $A(\langle s \rangle)$ must be defined for all s. Since the set $\{\langle s \rangle : s \in \mathbb{S}\}$ is a *maximal* prefix-free set, it follows that $A(p)$ must be undefined for all input strings p *not* of the form $\langle s \rangle$,

Proposition 6.1. *1. The empty set $\emptyset$ and $\mathbb{S}$ are recursive.*

2. If $R, S \subset \mathbb{S}$ are recursive then so are $R \cup S$ and $R \cap S$.

3. If S is recursive then so is its complement $\bar{S} = \mathbb{S} \setminus S$.

Proof ▶. 1. This is trivial.

2. Suppose A, B are acceptors for S, T Then we construct a machine C which first emulates A, and then emulates B.

   More precisely, given input $\langle s \rangle$, C first saves the input, and then emulates A, taking $\langle s \rangle$ as input.

We know that A will end by outputting 0 or 1.

If A ends by outputting 0, then C outputs nothing, but instead emulates B, again with input $\langle s \rangle$. If A ends by outputting 1, then C outputs 1 and halts.

Evidently C accepts the union $A \cup B$.

We construct a machine which accepts the intersection $A \cap B$ in exactly the same way, except that now it halts if A outputs 0, and emulates B if A outputs 1.

3. Suppose A accepts S. Let the machine C be identical to A, except that C outputs 1 when A outputs 0, and 0 when A outputs 1. Then C accepts the complementary set $\bar{S}$

6.1.1 Recursive codes

We say that a code

$$\gamma : X \to \mathbb{S}$$

for a set X is *recursive* if the image $\mathrm{im}(\gamma) \subset \mathbb{S}$ is recursive.

For example, the codes $\langle n \rangle$ and $\langle s \rangle$ that we have used for numbers and strings are both recursive and prefix-free (since we want to use them as input to Turing machines): and the same is true of our code $\langle T \rangle$ for Turing machines, and indeed all other codes we have used.

6.2 Recursively enumerable sets

Definition 6.2. *The set $S \subset \mathbb{S}$ is said to be* recursively enumerable *if there exists a Turing machine T such that*

$$s \in S \iff s = T(\langle p \rangle) \text{ for some } p \in \mathbb{S}.$$

We will say in such a case that the machine T *outputs* S.

Proposition 6.2. *1. A recursive set is recursively enumerable.*

2. A set $S \subset \mathbb{S}$ is recursive if and only if S and its complement $\mathbb{S} \setminus S$ are both recursively enumerable.

Proof ▶. 1. Suppose S is recursive. Let A be an acceptor for S. Then a slight modification A' of A will output S. Thus given an input string $\langle s \rangle$, A' first saves $\langle s \rangle$ and then emulates A taking $\langle p \rangle$ as input. If A concludes by outputting 1 then A' outputs s; while if A concludes by outputting 0 then A' goes into an infinite loop.

2. If S is recursive then so is its complement $\bar{S} = \mathbb{S} \setminus S$, by Proposition 6.1 So if S is recursive then both S and $\bar{S}$ are recursively enumerable.

   Conversely, suppose S and $\bar{S}$ are recursively enumerable. Let C, D output $S, \bar{S}$, respectively (always with coded input $\langle p \rangle$). Then we construct an acceptor A for S as follows.

   Given an input string $\langle p \rangle$, A starts by saving $\langle p \rangle$. Then A runs through a sequence of steps, which we will call Stage 1, Stage 2, ... At stage n, A runs through all strings p of length $\leq n$, carrying out n steps in the computation of $C(\langle p \rangle)$ and then n steps in the computation of $D(\langle p \rangle)$, saving the output string in coded form in either case. If one or both computations end then the output is compared with $\langle s \rangle$. If $C(\langle p \rangle) = \langle s \rangle$ then A outputs 1 and halts; if $D(\langle p \rangle) = \langle s \rangle$ then A outputs 0 and halts.

   One or other event must happen sooner or later since C and D together output all strings $s \in \mathbb{S}$.

This trick is reminiscent of the proof that $\mathbb{N} \times \mathbb{N}$ is enumerable, where we arrange the pairs (m, n) in a 2-dimensional array, and then run down the diagonals,

$$(0,0), (0,1), (1,0), (0,2), (1,1), (2,0), (0,3), (1,2), \ldots.$$

So we will call it the 'diagonal trick'.

It should not be confused with Cantor's entirely different and much more subtle 'diagonal method', used to show that $\#(X) < \#(2^X)$ and in the proof of the Halting Theorem. Note that Cantor's method is used to prove that something is *not* possible, while the diagonal trick is a way of showing that some procedure *is* possible.

Proposition 6.3. *1. $\emptyset, \mathbb{S}$ are recursively enumerable.*

2. If $R, S \subset \mathbb{S}$ are recursively enumerable then so are $R \cup S$ and $R \cap S$.

Proof ►. 1. This follows at once from the fact that $\emptyset$ and $\mathbb{S}$ are recursive.

2. Suppose C, D output R, S. In each case we use the diagonal trick; at stage n we input $\langle p \rangle$ for all p of length $\leq n$, and run C and D for n steps, and determine for which p (if any) C or D halts.

   For $R \cup S$ we simply output $C(\langle p \rangle)$ or $D(\langle p \rangle)$ in each such case.

   For $R \cap S$, we check to see if $C(\langle p \rangle) = D(\langle p' \rangle) = s$ for any inputs p, p', and if there are any such we output s.

6.3 The main theorem

Theorem 6.1. *There exists a set $S \subset \mathbb{S}$ which is recursively enumerable but not recursive.*

Proof ▶. Suppose U is a universal machine. By the Halting Theorem 5.1,

$$S = \{p : U(\langle p \rangle) \text{ defined}\}$$

is not recursive.

For a halting machine in this case is precisely an acceptor for S; and we saw that such a machine cannot exist.

It is easy to see that S is recursively enumerable, using the diagonal trick. At stage n we run though strings p of length $\leq n$, and follow the computation of $U(< p >)$ for n steps, If $U(< p >)$ completes in this time we output p.

It is clear that we will output all $p \in S$ sooner or later.

Chapter 7

Kraft's Inequality and its Converse

KRAFT'S INEQUALITY constrains entropy to increase at a certain rate. Its converse—sometimes known as Chaitin's lemma—shows that we can construct machines approaching arbitrarily close to this constraint.

7.1 Kraft's inequality

Recall that, for any Turing machine T, the set of strings

$$S = \{p \in \mathbb{S} : T(p) \text{ defined}\}$$

is *prefix-free*.

Theorem 7.1. *(Kraft's Inequality) If $S \subset \mathbb{S}$ is prefix-free then*

$$\sum_{s \in S} 2^{-|s|} \leq 1.$$

Proof ▶. To each string $s = b_1 b_2 \dots b_n$ we associate the binary number

$$B(s) = 0 \cdot b_1 b_2 \dots b_n \in [0, 1),$$

and the half-open interval

$$I(s) = [B(s), B(s) + 2^{-|s|}) \subset [0, 1).$$

Lemma 1. *The real numbers $B(s),\ s \in \mathbb{S}$ are dense in $[0, 1)$.*

Proof ▶. If

$$x = 0.b_1 b_2 \cdots \in [0, 1)$$

then

$$B(b_1),\ B(b_1 b_2),\ B(b_1 b_2 b_3) \to x.$$

Recall that we write $s \prec s'$ to mean that s is a prefix of s', eg

$$01101 \prec 0110110.$$

Lemma 2. *For any two strings $s, s' \in \mathbb{S}$,*

1. *$B(s') \in I(s) \iff s \prec s'$;*
2. *$I(s') \subset I(s) \iff s \prec s'$;*
3. *$I(s), I(s')$ are disjoint unless $s \prec s'$ or $s' \prec s$*

Proof ▶. 1. Let

$$s = b_1 \dots b_n.$$

Suppose $s \prec s'$, say

$$s' = b_1 \dots b_n b_{n+1} \dots b_{n+r}.$$

Then

$$B(s) \le B(s') = B(s) + 2^{-n} 0.b_{n+1} \dots b_{n+r} < B(s) + 2^{-n} = B(s) + 2^{-|s|}.$$

Conversely, supppose $s \not\prec s'$. Then either $s' \prec s$ (but $s' \neq s$); or else s, s' differ at some point, say

$$s = b_1 \dots b_{r-1} b_r b_{r+1} \dots b_n,\ s' = b_1 \dots b_{r-1} c_r c_{r+1} \dots c_m,$$

where $b_r \neq c_r$.

If $s' \prec s$ or $b_r = 1,\ c_r = 0$ then $B(s') < B(s)$.

If $b_r = 0,\ c_r = 1$ then

$$B(s') \ge 0.b_1 \dots b_{r-1} 1 > B(s) = 0.b_1 \dots b_{r-1} 0 b_{r+1} \dots b_n/$$

Thus

$$B(s) = \frac{a}{2^n},\ B(s') = \frac{b}{2^n},$$

with $a < b$. Hence

$$B(s') \ge B(s) + \frac{1}{2^n}.$$

2. Suppose $s \prec s'$. Then

$$B(s'') \in I(s') \Longrightarrow s' \prec s'' \Longrightarrow s \prec s'' \Longrightarrow B(s'') \in I(s).$$

It follows that

$$I(s') \subset I(s).$$

Conversely,

$$I(s') \subset I(s) \Longrightarrow B(s') \in I(s) \Longrightarrow s \prec s'.$$

3. If $I(s),\ I(s')$ are disjoint then we can find $s'' \in \mathbb{S}$ such that

$$B(s'') \in I(s) \cap I(s'),$$

so that

$$s \prec s'',\ s' \prec s''$$

which implies that

$$s \prec s' \text{ or } s' \prec s.$$

Conversely,

$$s \prec s' \Longrightarrow I(s') \subset I(s),\ s' \prec s \Longrightarrow I(s) \subset I(s');$$

and in neither case are $I(s),\ I(s')$ disjoint.

It follows from the last result that if the set of strings $S \subset \mathbb{S}$ is prefix-free then the half-intervals

$$\{I(s) : s \in S\}$$

are disjoint; and so, since they are all contained in $[0, 1)$,

$$\sum_{s \in S} |I(s)| = \sum_{s \in S} 2^{-|s|} \leq 1.$$

7.1.1 Consequences of Kraft's inequality

Proposition 7.1. *For each Turing machine T,*

$$\sum_{s \in \mathbb{S}} 2^{-H_T(s)} \leq 1.$$

Proof ▶. We know that

$$\sum_{p:T(p) \text{ is defined}} 2^{-|p|} \leq 1.$$

But each s for which $T(s)$ is defined arises from a unique minimal input

$$p = \mu_T(s);$$

while if $T(s)$ is not defined that

$$H_T(s) = \infty \Longrightarrow 2^{-H_T(s)} = 0.$$

It follows that the entropy of strings must increase sufficiently fast to ensure that

$$\sum_{s \in \mathbb{S}} 2^{-H(s)} \leq 1.$$

Thus there cannot be more than 2 strings of entropy 2, or more than 16 strings of entropy 4; if there is one string of entropy 2 there cannot be more than 2 of entropy 3; and so on.

7.2 The converse of Kraft's inequality

Theorem 7.2. *Suppose $\{h_i\}$ is a set of integers such that*

$$\sum 2^{-h_i} \leq 1.$$

Then we can find a prefix-free set $\{p_i\} \subset \mathbb{S}$ of strings such that

$$|p_i| = h_i.$$

Moreover this can be achieved by the following strategy: The strings $p_0, p_1, \ldots$ are chosen successively, taking p_i to be the first string (in lexicographical order) of length h_i such that the set

$$\{p_0, p_1, \ldots, p_i\}$$

is prefix-free.

Recall that the lexicographical order of $\mathbb{S}$ is

$$\square < 0 < 1 < 00 < 01 < 10 < 11 < 000 < \cdots,$$

where $\square$ denotes the empty string.

Proof ▶. Suppose the strings $p_0, p_1, \ldots, p_{i-1}$ have been chosen in accordance with the above specification. The remaining space (the 'gaps' in $[0,1)$)

$$G = [0,1) \setminus (I(p_0) \cup I(p_1) \cup \cdots \cup I(p_{i-1}))$$

is expressible as a finite union of disjoint half-open intervals $I(s)$, say

$$C = I(s_0) \cup I(s_1) \cup \cdots \cup I(s_j)$$

where

$$B(s_0) < B(s_1) < \cdots < B(s_j).$$

(This expression is unique if we agree to amalgamate any adjoining 'twin' intervals of the form

$$B(b_1, \ldots, b_r, 0),\ B(b_1, \ldots, b_r, 1)$$

to form the single interval

$$B(b_1, \ldots, b_r)$$

of twice the length.)

Lemma 3. *The intervals $I(s_0), \ldots, I(s_j)$ are strictly increasing in length, ie*

$$|s_0| > |s_1| > \cdots > |s_j|;$$

and

$$h_i \leq |s_j|,$$

so that it is possible to add another string p_i of length h_i.

Proof ▶. We prove the result by induction on i. Suppose it is true for the prefix-free set $\{p_0, \ldots, p_{i-1}\}$.

Since the intervals $I(s_k)$ are strictly increasing in size, each $I(s_k)$ is at most half as large as its successor $I(s_{k+1})$:

$$|I(s_k)| \leq \frac{1}{2}|I(s_{k+1}|.$$

It follows that the total space remaining is

$$< |I(s_j)| \left(1 + \frac{1}{2} + \frac{1}{4} + \cdots\right) = 2|I(s_j)|.$$

The next interval we are to add is to have length h_i. By hypothesis

$$2^{-h_0} + \cdots + 2^{h_{i-1}} + 2^{h_i} \leq 1.$$

Thus

$$\begin{aligned} 2^{-h_i} &\leq 1 - 2^{h_0} - \cdots - 2^{h_{i-1}} \\ &= |[0,1) \setminus (I(p_0) \cup I(p_1) \cup \cdots \cup I(p_{i-1}))| \\ &= |I(s_0) \cup I(s_1) \cup \cdots \cup I(s_j)| \\ &< 2|I(s_j)|. \end{aligned}$$

It follows that

$$2^{-h_i} \leq |I(s_j)|,$$

or

$$h_i \geq |s_j|.$$

So we can certainly fit an interval $I(p)$ of length 2^{-h_i} into one of our 'gap' intervals $I(s_k)$.

By prescription, we must take the 'first' position available for this new interval. Let us determine where 2^{-h_i} first fits into the sequence of strictly increasing gaps $I(s_0), I(s_1), \ldots$. Suppose

$$|I(s_{k-1})| < 2^{-h_i} \leq |I(s_k)|.$$

Then $I(s_k)$ is the first 'gap' into which we can fit an interval $I(p)$ of length 2^{-h_i}.

If in fact

$$2^{-h_i} = |I(s_k)|$$

then we set

$$p_i = s_k.$$

In this case, the gap is completely filled, and we continue with one fewer gap, the remaining gaps evidently satisfying the conditions of the lemma.

If however

$$2^{-h_i} < |I(s_k)|$$

then our strategy prescribes that $I(p_i)$ is to come at the 'beginning' of $I(s_k)$, ie

$$p_i = s_k \overbrace{0 \ldots 0}^{e \text{ 0's}},$$

where

$$e = h_i - |s_k|.$$

We note that

$$I(s_k) \setminus I(p_i) = I(t_0) \cup I(t_1) \cup \cdots \cup I(t_{e-1}),$$

where

$$t_0 = s_k \overbrace{0\ldots0}^{e-1\ 0\text{'s}} 1,\ t_1 = s_k \overbrace{0\ldots0}^{e-2\ 0\text{'s}} 1,\ \ldots, t_{e-2} = s_k 01,\ t_{e-1} = s_k 1.$$

Thus after the addition of the new interval $I(p_k)$ the complement

$$[0,1) \setminus (I(p_0) \cup \cdots \cup I(p_i)) = \\ I(s_0) \cup \cdots \cup I(s_{k-1}) \cup I(t_0) \cup \cdots I(t_r) \cup I(s_{k+1}) \cup \cdots \cup I(s_j)$$

retains the property described in the lemma. It therefore follows by induction that this property always holds.

It follows that the strategy can be continued indefinitely, creating a prefix-free set of strings with the required properties.

7.3 Chaitin's lemma

We would like to construct a machine T so that specified strings $s_0, s_1, \ldots$ have specified entropies $h_0, h_1, \ldots$:

$$H_T(s_i) = h_i.$$

By Kraft's Inequality this is certainly not possible unless

$$\sum_i 2^{-h_i} \leq 1.$$

But suppose that is so. The converse to Kraft's inequality encourages us to believe that we should be able to construct such a machine.

But one question remains. What exactly do we mean by saying that the entropies h_i are 'specified'? *How* are they specified?

If the machine T is to 'understand' the specification, it must be in 'machine-readable' form. In other words, we must have *another* machine M outputting the numbers h_i.

Theorem 7.3. *Suppose*

$$S \subset \mathbb{S} \times \mathbb{N}$$

is a set of pairs (s, h_s) such that

1. *The integers h_s satisfy Kraft's condition:*

$$\sum_{(s,h_s)\in S} 2^{-h_s} \leq 1.$$

2. The set S is recursively enumerable.

Then there exists a Turing machine T such that

$$H_T(s) \leq h_s$$

for all $(s, h_s) \in S$.

Proof ▶. By definition, there exists a machine M which generates the set S, say

$$M(n) = (s_n, h_n) \in S.$$

Suppose we are given an input string p. We have to determine $T(p)$. Our machine does this by successively building up a prefix-free set

$$P = \{p_0, p_1, p_2, \dots\},$$

where $|p_n| = h_n$, according to the prescription above. As each p_i is created, it is compared with the given string p; and if $p_n = p$ then T outputs the string s_n and halts.

If p never occurs in the prefix-free set P then $T(p)$ is undefined.

More fully, T functions in stages $0, 1, 2, \dots$. At stage n, T emulating each of $M(0), M(1), \dots, M(n)$ for n steps.

If $M(r)$ halts after $m \leq N$ steps, with

$$M(r) = \langle s_r \rangle \langle h_r \rangle.$$

Then T adds a further string p_i with $|p_i| = h_r$ to the prefix-free set

$$P = \{p_0, p_1, \dots, p_{i-1}\}$$

which it is building up, by following the 'Kraft prescription'.

Summary

We have constructed a machine T with specified entropies $H_T(s_i)$ for specified the string s_i, provided these entropies satisfy Kraft's inequality, and can be recursively generated.

Chapter 8

A Statistical Interpretation of Algorithmic Entropy

We can regard a Turing machine as a kind of random generator, with a certain probability of outputting a given string. This suggests an alternative definition of the entropy of a string, more in line with the concepts of Shannon's statistical information theory. Fortunately, we are able to establish the equivalence of this new definition with our earlier one, at least up to an additive constant.

8.1 Statistical Algorithmic Entropy

Definition 8.1. *The statistical algorithmic entropy $h_T(s)$ of the string s with respect to the Turing machine T is defined by*

$$2^{-h_T(s)} = \sum_{p:T(p)=s} 2^{-|p|},$$

that is,

$$h_T(s) = -\lg\left(\sum_{p:T(p)=s} 2^{-|p|}\right).$$

We set

$$h(s) = h_U(s),$$

where U is our chosen universal machine.

Recall the convention that $\lg x$ denotes $\log_2 x$; all our logarithms will be taken to base 2.

Proposition 8.1. *For any Turing machine T, and any string s,*

$$h_T(s) \leq H_T(s);$$

and in particular,

$$h(s) \leq H(s).$$

Proof ▶. If T never outputs s then the sum is empty and so has value 0, giving

$$h_T(s) = \infty = H_T(s).$$

Otherwise, one of the programs that outputs s is the minimal program (for T) $p = \mu_T(s)$. Thus

$$2^{-h_T(s)} = \sum_{p:T(p)=s} 2^{-\|p\|} \geq 2^{-\|\mu_T(s)\|} = 2^{-H_T(s)},$$

and so

$$h_T(s) \leq H_T(s).$$

Proposition 8.2.

$$h_T(s) \geq 0.$$

Proof ▶. Since

$$\{p : T(p) = s\} \subset \{p : T(p) \text{ defined}\},$$

it follows that

$$\sum_{p:T(p)=s} 2^{-|p|} \leq \sum_{p:T(p) \text{ defined}} 2^{-|p|} \leq 1,$$

by Kraft's Inequality 7.1.

Hence

$$h_T(s) = -\lg \sum_{p:T(p)=s} 2^{-|p|} \geq 0.$$

Proposition 8.3. *For any string $s \in \mathbb{S}$.*

$$h(s) < \infty.$$

Proof ▶. We can certainly construct a machine T which outputs a given string s without any input. Then

$$U(\langle T \rangle) = s.$$

Hence

$$H(s) \leq |\langle T \rangle| < \infty,$$

and *a fortiori*

$$h(s) \leq H(s) < \infty.$$

8.2 The Turing machine as random generator

Imagine the following scenario. Suppose we choose the input to T by successively tossing a coin. (We might employ a mad statistician for this purpose.) Thus if the current rule requires that T should read in an input bit, then a coin is tossed, and 1 or 0 is input according as the coin comes up heads or tails. The experiment ends if and when the machine halts.

The expectation that this results in a given string s being output is

$$\mathrm{P}_T(s) = \sum_{p:T(p)=s} 2^{-|s|}.$$

Recall that in Shannon's Information Theory, if an event e has probability p of occurring, then we regard the occurrence of e as conveying $-\lg p$ bits of information. (Thus occurrence of an unlikely event conveys more information than occurrence of an event that was more or less inevitable.)

In our case, the 'event' is the outputting of the string s. So the information conveyed by the string is just

$$h_T(s) = -\lg \mathrm{P}_T(s).$$

Summary

The statistical algorithmic entropy $h(s)$ gives us an alternative measure of the informational content of a string. Fortunately we shall be able to establish that it is equivalent to our earlier measure, up to the ubiquitous constant:

$$h(s) = H(s) + O(1).$$

Chapter 9

Equivalence of the Two Entropies

WE SHOW that the rival definitions of algorithmic entropy, $H(s)$ and $h(s)$, are in fact equivalent.

Theorem 9.1.

$$h(s) = H(s) + O(1).$$

More precisely, there exists a contant C independent of s such that

$$h(s) \leq H(s) \leq h(s) + C$$

for all $s \in \mathbb{S}$.

Proof ▶. As we saw in Proposition 8.1,

$$h(s) \leq H(s).$$

We must show that there exists a constant C, dependent only on our choice of universal machine U, such that

$$H(s) \leq h(s) + C$$

for all strings $s \in \mathbb{S}$.

Lemma 1.

$$\sum_{s \in \mathbb{S}} 2^{-h_T(s)} \leq 1.$$

Proof ▶. Each p for which $T(p)$ is defined contributes to $h_T(s)$ for just one s. Hence

$$\begin{aligned}\sum_{s\in\mathbb{S}} 2^{-h_T(s)} &= \sum_{s\in\mathbb{S}} \left(\sum_{p:T(p)=s} 2^{-|s|} \right) \\ &= \sum_{p:T(p)\text{ defined}} 2^{-|s|} \\ &\leq 1,\end{aligned}$$

since the set of p for which $T(p)$ is defined is prefix-free.

Thus the numbers $h(s)$ satisfy Kraft's Inequality. However, we cannot apply the converse as it stands since these numbers are not in general integral.

We therefore set

$$h_s = [h(s)] + 1$$

for each string $s \in \mathbb{S}$. (Here $[x]$ denotes, as usual, the greatest integer $\leq x$.) Thus

$$h(s) < h_s \leq h(s) + 1.$$

Since

$$\sum 2^{-h_s} \leq \sum 2^{-h(s)} \leq 1,$$

the integers h_s, or rather the set of pairs

$$S = \{(s, h_s)\} \in \mathbb{S} \times \mathbb{N},$$

satisfy the first criterion of Chaitin's Lemma.

The Converse, if we could apply it, would allow us to construct a machine M such that

$$H_M(s) \leq h_s$$

for all s with $h_s < \infty$. It would follow from this that

$$\begin{aligned}H(s) &\leq H_M(s) + |\langle M \rangle| \\ &\leq h_s + O(1) \\ &\leq h(s) + O(1).\end{aligned}$$

Unfortunately, *we have no reason to suppose that the h_s are recursively enumerable*. We cannot therefore apply the Converse directly, since we have not shown that its second criterion is fulfilled.

Fortunately, a nimble side-step steers us round this obstacle.

Lemma 2. *Suppose T is a Turing machine. Then the set*

$$S' = \{(s, m) \in \mathbb{S} \times \mathbb{N} : h_T(s) > 2^{-m}\}$$

is recursively enumerable.

Proof ▶. We construct a machine M which runs as follows.

At the nth stage, M runs through all $2^{n+1} - 1$ strings p of length $\leq n$. For each such string p, M emulates T for n steps. If T halts within these n steps, with $s = T(p)$, a note is made of the pair $(s, |p|)$.

At the end of the nth stage, the accumulated total

$$\mathrm{P}'_T(s) = \sum_{|p| \leq n : T(p) = s} 2^{-|p|}$$

is calculated for each string s that has appeared; and for each new integer $m = m(s)$ for which

$$\mathrm{P}'_T(s) \geq 2^{-m}$$

the pair (s, m) is output.

(Note that as more inputs are considered, $\mathrm{P}'_T(s)$ is increasing, tending towards $\mathrm{P}_T(s)$. Thus m is decreasing, passing through integers $\geq h_T(s)$.)

Lemma 3. *With the notation of the last lemma,*

$$\sum_{s,m:(s,m) \in S'} 2^{-m} \leq 2.$$

Proof ▶. As we saw in the proof of the last Lemma, the $m = m(s)$ that arise for a given s are $\geq h_T(s)$. Hence their sum is

$$< h_T(s) \left(1 + \frac{1}{2} + \frac{1}{2^2} + \cdots\right) = 2h_T(s).$$

Thus the sum for all s is

$$< 2 \sum_s h_T(s) \leq 2,$$

by Lemma 1.

Now we *can* apply the Converse to the set

$$S'' = \{(s, m+1) : (s, m) \in S'\};$$

for we have shown in Lemma 3 that this set satisfies the first criterion, while we saw in Lemma 1 that it is recursively enumerable.

Thus we can construct a machine M with the property that for each $(s, m) \in S$ we can find a program p such that

$$M(p) = s, \quad |p| \leq h_s + 1.$$

It follows that

$$H_M(s) \leq h_s + 1;$$

and so, taking $T = U$,

$$\begin{aligned} H(s) &\leq H_M(s) + |\langle M \rangle| \\ &\leq h_s + |\langle M \rangle| \\ &= h_s + O(1) \\ &\leq h(s) + O(1). \end{aligned}$$

Summary

We have established that $H(s)$ and $h(s)$ are equivalent definitions of entropy. It is thus open to us to use whichever is more convenient for the problem in hand.

Chapter 10

Conditional entropy re-visited

RECALL THAT in Chapter 4 we defined the conditional entropy $H(s \mid t)$ of one string s given another string t. We stated, but did not prove, the fundamental identity

$$H(s,t) = H(t) + H(s \mid t).$$

Informally, this says that the information in t, together with the additional information in s, is precisely the information in both s and t. This result is the last piece of the jigsaw in the basic theory of algorithmic entropy. The proof we give now is somewhat convoluted, involving the equivalence of the two entropies, as well as a further application of the converse to Kraft's Inequality.

10.1 Conditional Entropy

Recall the definition of $H(s \mid t)$: Let $\tau = \mu(t)$ be the minimal input outputting t from our universal machine U. Then we consider the set of pairs (T, p) consisting of a Turing Machine T and a string p such that

$$T(\tau p) = s;$$

and we define $H(s \mid t)$ to be the minimum of

$$|\langle T \rangle| + |p|.$$

(We cannot simply take the minimum of $|p|$ over strings p such that

$$p : U(\tau p) = s,$$

because U is going to halt as soon as it has read in τ. So we make an indirect reference to the fact that

$$H(s) \leq H_T(s) + |\langle T \rangle|,$$

since $U(\langle T \rangle p) = T(p)$.)

10.2 The last piece of the jigsaw

Theorem 10.1.

$$H(s,t) = H(t) + H(s \mid t) + O(1).$$

Our proof is in two parts.

1. The easy part:

$$H(s,t) \leq H(t) + H(s \mid t) + O(1).$$

2. The harder part:

$$H(s \mid t) \leq H(s,t) - H(t) + C,$$

where C is a constant depending only on our choice of universal machine U.

10.3 The easy part

Proposition 10.1. *There exists a Turing machine M such that*

$$H_M(s,t) \leq H(t) + H(s \mid t)$$

for all $s, t \in \mathbb{S}$.

Proof ▶. Let

$$\tau = \mu(t)$$

be the minimal program for t:

$$U(\tau) = t, \quad H(t) = |\tau|.$$

By the definition of conditional entropy $H(s \mid t)$, we can find a machine T and a string p such that

$$T(\tau p) = s, \quad H(s \mid t) = |\langle T \rangle| + |p|.$$

It follows that

$$U(\langle T\rangle \tau p) = T(\tau p) = t.$$

Now we construct the machine M as follows. M starts by imitating U, so that if the input string is

$$q = \langle T\rangle \tau p$$

then M will compute $U(q) = t$. However, it does not output t but simply records its value for later use.

But now M goes back to the beginning of the string q (which it has wisely stored) and skips the machine code $\langle T\rangle$.

Now M imitates U again, but this time with input τp. Since $U(\tau) = s$, U will only read in the prefix τ before halting and outputting s. Our machine M does not of course output s. Instead it outputs the coded version $\langle s\rangle$.

Finally, it goes back to the remembered string t and outputs $\langle t\rangle$ before halting.

In summary,

$$M(\langle T\rangle \tau p) = \langle s\rangle\langle t\rangle.$$

It follows that

$$\begin{aligned} H_M(s,t) &\le |\langle T\rangle| + |\tau| + |p| \\ &= |\tau| + (|\langle T\rangle| + |p|) \\ &= H(t) + H(s \mid t). \end{aligned}$$

Corollary 10.1. *We have*

$$H(s,t) \le H(t) + H(s \mid t) + O(1).$$

Proof ▶. Since

$$H(s,t) \le H_M(s,t) + |\langle M\rangle|,$$

the Proposition implies that

$$\begin{aligned} H(s,t) &\le H(t) + H(s \mid t) + |\langle M\rangle| \\ &= H(t) + H(s \mid t) + O(1). \end{aligned}$$

10.4 The hard part

Proposition 10.2. *For each string $t \in \mathbb{S}$,*

$$\sum_{s\in\mathbb{S}} 2^{-(h(s,t)-h(t))} \le C,$$

where C is a constant depending only on our choice of universal machine U.

Proof ▶. **Lemma 1.** *Given a machine T, there exists a machine M such that*

$$\sum_{s\in\mathbb{S}} \mathrm{P}_T(s,t) \leq \mathrm{P}_M(t)$$

(where $\mathrm{P}_T(s,t) = \mathrm{P}_T(\langle s\rangle\langle t\rangle)$).

Proof ▶. Let the machine M start by imitating T, except that instead of outputting $\langle s\rangle\langle t\rangle$, it skips $\langle s\rangle$ and then decodes $\langle t\rangle$ —that is, as T outputs $\langle s\rangle\langle t\rangle$ M outputs t, and then halts.

It follows that

$$T(p) = \langle s\rangle\langle t\rangle \Longrightarrow M(p) = t.$$

Hence

$$\begin{aligned}\bigcup_{s\in\mathbb{S}} \mathrm{P}_T(s,t) &= \sum_{s\in\mathbb{S}} \left(\sum_{p:T(p)=\langle s\rangle\langle t\rangle} 2^{-|p|} \right) \\ &\leq \sum_{p:M(p)=t} 2^{-|p|} \\ &= P_M(t).\end{aligned}$$

Lemma 2. *With the same assumptions as the last Lemma,*

$$\sum_{s\in\mathbb{S}} 2^{-h_T(s,t)} \leq 2^{-h_M(t)}.$$

Proof ▶. This follows at once from the last Lemma, since

$$2^{-h_T(s,t)} = \mathrm{P}_T(s,t), \quad 2^{-h_M(t)} = \mathrm{P}_M(t).$$

Lemma 3. *For any Turing machine T,*

$$h_T(s) \leq 2^{|\langle T\rangle|} h(s).$$

Proof ▶. Since

$$T(p) = s \Longleftrightarrow U(\langle T\rangle p) = s,$$

it follows that

$$\begin{aligned}2^{-h(s)} &= \sum_{q:U(q)=s} 2^{-|q|} \\ &\geq \sum_{p:T(p)=s} 2^{-|\langle T\rangle p|} \\ &= 2^{-|\langle T\rangle|} \sum_{p:T(p)=s} 2^{-|p|} \\ &= 2^{-|\langle T\rangle|}\ h(s).\end{aligned}$$

Lemma 4.

$$\sum_{s\in\mathbb{S}} 2^{-h(s,t)} \leq 2^{-h(t)+c},$$

where c is a constant depending only on our choice of universal machine U.

Proof ▶. This follows at once on taking $T = U$ in Lemma 2, and applying Lemma 3 with $T = M$.

The result follows on taking $h(t)$ to the other side.

Corollary 10.1.

$$\sum_{s\in\mathbb{S}} 2^{-(H(s,t)-H(t))} \leq C',$$

where C' depends only U.

Proof ▶. This follows from the Proposition on applying the Equivalence Theorem 9.1.

We can now formulate our strategy. Let us fix the string t. Suppose

$$\tau = \mu(t)$$

is the minimal program for t:

$$U(\tau) = s, \quad |\tau| = H(t).$$

We can re-write Corollary 10.1 as

$$\sum_{s\in\mathbb{S}} 2^{-(H(s,t)-H(t)+c)} \leq 1,$$

where c depends only on U. Thus the numbers

$$h_s = H(s,t) - H(t) + c$$

satisfy Kraft's inequality

$$\sum_{s\in\mathbb{S}} 2^{-h_s} \leq 1.$$

If these numbers were integers, and were recursively enumerable, then we could find a prefix-free set (depending on t)

$$P_t = \{p_{st} : s \in \mathbb{S}\}$$

such that

$$|p_{st}| \leq H(s,t) - H(t) + c.$$

Now let us prefix this set with $\mu(t) = \tau$:

$$\mu(t)P_t = \{\mu(t)p_{st} : s \in \mathbb{S}\}.$$

It is easy to see that the sets $\mu(t)P_t$ for different t are disjoint; and their union

$$P = \bigcup_{t\in\mathbb{S}} \mu(t)P_t = \{s, t \in \mathbb{S} : \mu(t)p_{st}\}.$$

is prefix-free.

Thus we may—with luck—be able to construct a machine T such that

$$T(\mu(t)p_{st}) = t$$

for each pair $s, t \in \mathbb{S}$.

From this we would deduce that

$$\begin{aligned} H(s \mid t) &\le |p_{st}| + |\langle T\rangle| \\ &\le H(s,t) - H(t) + c + |\langle T\rangle| \\ &= H(s,t) - H(t) + O(1), \end{aligned}$$

as required.

Now for the gory details. Our machine T starts by imitating U. Thus if the input string is

$$q = \tau p \quad (\tau \in \Omega(U))$$

M begines by reading in τ and computing

$$t = U(\tau).$$

However, instead of outputting t, T stores τ and t for further use.

We are only interested in the case where τ is the *minimal* program for t:

$$\tau = \mu(t), \quad |\tau| = H(t).$$

Of course, this is not generally true. But if it is not true then we do not care whether $T(q)$ is defined or not, or if it is defined what value it takes. Therefore we assume in what follows that $\tau = \mu(s)$.

By Corollary 10.1 above,

$$\sum_{s\in\mathbb{S}} 2^{-(H(s,t)-H(t)+c)} \le 1$$

Thus if we set

$$h_s = [H(s,t) - H(t) + c + 2]$$

then

$$\sum_{s\in\mathbb{S}} 2^{-h_s} \leq \frac{1}{2}.$$

As in Chapter 9, we cannot be sure that the set

$$S' = \{(s, h_s) : s \in \mathbb{S}\} \subset \mathbb{S} \times \mathbb{N}$$

is recursively enumerable. However, we can show that a slightly—but not too much—larger set S *is* recursively enumerable. (This is the reason for introducing the factor $\frac{1}{2}$ above—it allows for the difference between S and S'.)

Lemma 5. *Let*

$$S' = \{(s, h_s) : s \in \mathbb{S}\}, \quad S'' = \{(s, n) : n \geq h_s\}.$$

There exists a recursively generated set $S \subset \mathbb{S} \times \mathbb{N}$ such that

$$S' \subset S \subset S''.$$

Proof ▶. We construct an auxiliary machine M which recursively generates the set

$$\Omega(U) = \{U(p) : p \in \mathbb{S}\}.$$

As each $U(p)$ is generated, M determines if it is of the form $\langle s\rangle\langle t\rangle$ (where t is the string $U(\tau)$). If it is then M checks if the pair (s, n), where

$$n = |p| - |\tau| + c + 1$$

has already been output; if not, it outputs $\langle s\rangle\langle n\rangle$.

Since

$$|p| \geq H(s, t)$$

by definition, while by hyothesis

$$|\tau| = H(t),$$

it follows that

$$n \geq H(s, t) - H(t) + c + 1 = h_s,$$

and so $(s, n) \subset S''$. Hence

$$S \subset S''.$$

On the other hand, with the particular input $p = \mu(s, t)$,

$$|p| = H(s, t)$$

and so $n = h_s$. Thus $(s, h_t) \in S$. and so

$$S' \subset S.$$

But now

$$\sum_{(t,n)\in\mathbb{S}} 2^{-n} \leq 1,$$

since each t contributes at most

$$2^{-h_t} + 2^{-h_t-1} + 2^{-h_t-2} + \cdots = 2 \cdot 2^{-h_t}.$$

We can therefore follow Kraft's prescription for a prefix-free set. As each pair (s, n) is generated, T determines a string p_{sn} such that

$$|p_{sn}| \leq 2^{-n},$$

in such a way that the set

$$P = \{p_{sn} : (s, n) \in S\}$$

is prefix-free.

As each string p_{sn} is generated, T checks the input string q to see if

$$q = \tau p_{sn}.$$

If this is true then the computation is complete: T outputs s and halts:

$$T(\tau p_{tn}) = s.$$

One small point: in comparing τp_{sn} with the input string q, T might well go past the end of q, so that $T(q)$ is undefined. However, in that case q is certainly not of the form $\tau p_{s'n'}$, since this would imply that

$$\tau p_{s'n'} \subset \tau p_{st},$$

contradicting the 'prefix-freedom' of P.

To recap, we have constructed a machine T such that for each pair $s, t \in \mathbb{S}$ we can find a string p_{st} with

$$T(\mu(t)p_{st}) = s, \quad |p_{st}| \leq H(s, t) - H(t) + c.$$

But now, from the definition of the conditional entropy $H(s \mid t)$,

$$\begin{aligned} H(s \mid t) &\leq |p_{st}| + |\langle T \rangle| \\ &\leq H(s, t) - H(t) + c + |\langle T \rangle| \\ &= H(s, t) - H(t) + O(1). \end{aligned}$$

Summary

With the proof that

$$H(s, t) = H(t) + H(s \mid t) + O(1)$$

the basic theory of algorithmic entropy is complete.

Appendix A

Cardinality

Cardinality — that is, Cantor's theory of infinite cardinal numbers — does not play a direct rôle in algorithmic information theory, or more generally in the study of computability, since all the sets that arise there are *enumerable*. However, the proof of Cantor's Theorem below, using Cantor's 'diagonal method', is the predecessor, or model, for our proof of the Halting Theorem and other results in Algorithmic Information Theory. The idea also lies behind Gödel's Unprovability Theorem, that in any non-trivial axiomatic system there are propositions that can neither be proved nor disproved.

A.1 Cardinality

Definition A.1. *Two sets X and Y are said to have the same cardinality, and we write*

$$\#(X) = \#(Y),$$

if there exists a bijection $f : X \to Y$.

When we use the '=' sign for a relation in this way we should verify that the relation is reflexive, symmetric and transitive. In this case that is trivial.

Proposition A.1. *1.* $\#(X) = \#(X)$*;*

2. $\#(X) = \#(Y) \Longrightarrow \#(Y) = \#(X)$*;*

3. $\#(X) = \#(Y)$ $\#(Y) = \#(Z) \Longrightarrow \#(X) = \#(Z)$.

By convention, we write

$$\#(\mathbb{N}) = \aleph_0.$$

Definition A.2. *We say that the cardinality of X is less than or equal to the cardinality of Y, and we write*

$$\#(X) \leq \#(Y),$$

if there exists a injection $f : X \to Y$.

We say that the cardinality of X is (strictly) less than the cardinality of Y, and we write

$$\#(X) < \#(Y),$$

if $\#(X) \leq \#(Y)$ and $\#(X) \neq \#(Y)$.

Proposition A.2. *1.* $\#(X) \leq \#(X)$*;*

2. $\#(X) \leq \#(Y)$ $\#(Y) \leq \#(Z) \Longrightarrow \#(X) \leq \#(Z)$.

Again, these follows at once from the definition of injectivity.

A.1.1 Cardinal arithmetic

Addition and multiplication of cardinal numbers is defined by

$$\#(X) + \#(Y) = \#(X + Y), \quad \#(X) \times \#(Y) = \#(X \times Y),$$

where $X + Y$ is the disjoint union of X and Y (ie if X and Y are not disjoint we take copies that are).

However, these operations are not very useful; for if one (or both) of $\aleph_1, \aleph_2$ are infinite then

$$\aleph_1 + \aleph_2 = \aleph_1 \aleph_2 = \max(\aleph_1, \aleph_1).$$

The power operation is more useful, as we shall see. Recall that 2^X denotes the set of subsets of X. We set

$$2^{\#(X)} = \#(2^X).$$

A.2 The Schröder-Bernstein Theorem

Theorem A.1.

$$\#(X) \leq \#(Y) \quad \#(Y) \leq \#(X) \Longrightarrow \#(X) = \#(Y).$$

Proof ▶. By definition there exist injective maps

$$f : X \to Y, \; g : Y \to X.$$

We have to construct a bijection

$$h : X \to Y.$$

To simplify the discussion, we assume that X and Y are disjoint (taking disjoint copies if necessary).

Given $x_0 \in X$, we construct the sequence

$$y_0 = f(x_0) \in Y,\ x_1 = g(y_0) \in X,\ y_1 = f(x_1) \in Y, \ldots.$$

There are two possibilities:

(i) The sequence continues indefinitely, giving a singly-infinite chain in X:

$$x_0, y_0, x_1, y_1, x_2, \ldots$$

(ii) There is a repetition, say

$$x_r = x_s$$

for some $r < s$. Since f and g are injective, it follows that the first repetition must be

$$x_0 = x_r,$$

so that we have a loop

$$x_0, y_0, x_1, y_1, \ldots, x_r, y_r, x_0.$$

In case (i), we may be able to extend the chain backwards, if $x_0 \in \mathrm{im}(g)$. In that case we set

$$x_0 = gy_{-1},$$

where y_{-1} is unique since g is injective.

Then we may be able to go further back:

$$y_{-1} = fx_{-1},\ x_{-2} = gy_{-1},\ \ldots.$$

There are three possibilities:

(A) The process continues indefinitely, giving a doubly-infinite chain

$$\ldots, x_{-n}, y_{-n}, x_{-n+1}, y_{-n+1}, \ldots, x_0, y_0, x_1, \ldots.$$

(B) The process ends at an element of X, giving a singly-infinite chain

$$x_{-n}, y_{-n}, x_{-n+1}, \ldots.$$

(C) The process ends at an element of Y, giving a singly-infinite chain

$$y_{-n}, x_{-n+1}, y_{-n+1}, \dots .$$

It is easy to see that these chains and loops are disjoint, partitioning the union $X + Y$ into disjoint sets. This allows us to define the map h on each chain and loop separately. Thus in the case of a doubly-infinite chain or a chain starting at an element $x_{-n} \in X$, or a loop, we set

$$hx_r = y_r;$$

while in the case of a chain starting at an element $y_{-n} \in Y$ we set

$$hx_r = y_{r-1}.$$

Putting these maps together gives a bijective map

$$h : X \to Y.$$

A.3 Cantor's Theorem

Theorem A.2. *The number of elements of a set is strictly less than the number of subsets of the set:*

$$\#(X) < \#(2^X).$$

Proof ▶. We have to show that $\#(X) \leq \#(2^X)$ but $\#(X) \neq \#(2^X)$.

There is an obvious injection $X \to 2^X$, namely

$$x \mapsto \{x\}.$$

Hence

$$\#(X) \leq \#(2^X).$$

Suppose there is a surjection

$$f : X \to 2^X.$$

Let

$$S = \{x \in X : x \notin f(x)\}.$$

Since f is surjective, there exists an element $s \in X$ such that

$$S = f(s).$$

We ask the question: *Does the element s belong to the subset S, or not?*

If $s \in S$, then from the definition of S,

$$s \notin f(s) = S.$$

On the other hand, if $s \notin S$, then again from the definition of S.

$$s \in S.$$

Either way, we encounter a contradiction. Hence our hypothesis is untenable: there is no surjection, and so no isomorphism, $f : X \to 2^X$, ie

$$\#(X) \neq \#(2^X).$$

A.4 Comparability

Our aim in this Section is to prove any 2 sets X, Y are *comparable*, ie

$$\text{either } \#(X) \leq \#(Y) \text{ or } \#(Y) \leq \#(X).$$

To this end we introduce the notion of well-ordering.

Recall that a *partial order* on a set X is a relation $\leq$ such that

1. $x \leq x$ for all x;
2. $x \leq y \;\; y \leq z \Longrightarrow x \leq z$;
3. $x \leq y \;\; y \leq x \Longrightarrow x = y$.

A partial order is said to be a *total order* if in addition, for all $x, y \in X$,

4. either $x \leq y$ or $y \leq x$.

A total order is said to be a *well-ordering* if

5. every non-empty subset $S \subset X$ has a least element $\mu(S) \in S$.

Examples:

1. The natural numbers $\mathbb{N}$ are well-ordered.
2. The integers $\mathbb{Z}$ are not well-ordered, since $\mathbb{Z}$ itself does not have a least element.
3. The set of positive reals $\mathbb{R}^+ = \{x \in \mathbb{R} : x \geq 0\}$ is not well-ordered, since the set $S = \{x > 0\}$ does not have a least element *in* S.

4. The set $N \times N$ with the lexicographic ordering

$$(m, n) \leq (m', n') \text{ if } m < m' \text{ or } m = m' \;\; n \leq n'$$

is well-ordered. To find the least element (m, n) in a subset $S \subset \mathbb{N} \times \mathbb{N}$ we first find the least m occuring in S; and then among the pairs $(m, n) \in S$ we find the least n.

5. The disjoint sum $\mathbb{N} + \mathbb{N}$, with the ordering under which every element of the first copy of $\mathbb{N}$ is less than every element of the second copy of $\mathbb{N}$, is well-ordered.

It follows at once from the definition that every subset $S \subset X$ of a well-ordered set is well-ordered.

A well-ordered set X has a first (least) element

$$x_0 = \mu(X).$$

Unless this is the only element, X has a second (next least) element

$$x_1 = \mu(X \setminus \{x_0\}).$$

Simitarly, unless these are the only elements, X has a third element

$$x_2 = \mu(X \setminus \{x_0, x_1\}),$$

and so on. Moreover after all these elements $x_0, x_1, x_2, \ldots$ (assuming they have not exhausted X) there is a next element

$$x_\omega = \mu(X \setminus \{x_0, x_1, x_2, \ldots\}).$$

Then comes the element

$$x_{\omega+1} = \mu(X \setminus \{x_0, x_1, x_2, \ldots, x_\omega\}),$$

and after that elements $x_{\omega+2}, x_{\omega+3}, \ldots$.

Proposition A.3. *There is at most one order-preserving isomorphism between 2 well-ordered sets.*

Proof ▶. Suppose

$$f, g : X \to Y$$

are 2 isomorphisms between the well-ordered sets X, Y. Let

$$z = \mu\left(\{x \in X : fx \neq gx\}\right).$$

In other words, z is the first point at which the maps f and g diverge.

We may assume without loss of generality that

$$fz < gz.$$

Since g is an isomorphism,

$$fz = gt$$

for some element $t \in X$. But now, since g is order-preserving,

$$gt < gz \Longrightarrow t < z \Longrightarrow gt = ft \Longrightarrow fz = ft \Longrightarrow z = t \Longrightarrow gz = gt = fy,$$

contrary to hypothesis. We conclude that $f = g$, ie $f(x) = g(x)$ for all $x \in X$.

Although we shall make no use of this, we associate an *ordinal number* (or just *ordinal*) to each well-ordered set. By the last Proposition, two well-ordered sets have the same ordinal number if and only if they are order-isomorphic.

A subset $I \subset X$ in a partially-ordered set X is called an *initial segment* if

$$x \in I \quad y \leq x \Longrightarrow y \in I.$$

It is easy to see that the set

$$I(x) = \{y \in X : y < x\}$$

(where $x < y$ means $x \leq y$ but $x \neq y$) is an initial segment in X for each $x \in X$.

In a well-ordered set *every initial subset $I \subset X$, except X itself, is of this form*; for it is easily seen that

$$I = I(x),$$

where

$$x = \mu(X \setminus I).$$

If an element x of a well-ordered set X is not the greatest element of X then it has an immediate successor x', namely

$$x' = \mu(\{y \in X : y > x\}).$$

But not every element $x \in X$ (apart from the minimal element x_0) need be a successor element. We call an element $x \neq x_0$ with no immediate predecessor a *limit element*.

Lemma 6. *If x is a limit element then*

$$I(x) = \cup_{y<x} I(y).$$

Proof ▶. Certainly

$$J = \cup_{y<x} I(y)$$

is an initial segment. Suppose

$$J = I(z),$$

where $z \leq x$.

If $z < x$ then x must be the immediate successor to z. For suppose $z < t < x$. Then

$$z \in I(t) \subset J = I(z),$$

contrary to the definition of the initial segment $I(z)$.

Lemma 7. *Suppose X, Y are well-ordered sets. Then X is order-isomorphic to at most one initial segment I of Y.*

Proof ▶. If X is isomorphic to two different initial segments $I, J \subset Y$ then there are two different order-preserving injective maps

$$f, g : X \to Y.$$

Suppose these maps diverge at $z \in X$:

$$z = \mu(\{x \in X : fx \neq gx\}).$$

We may assume without loss of generality that

$$fz < gz.$$

But then $fz \in J = \mathrm{im}(g)$, and so

$$fz = gt$$

for some $t \in X$. Thus

$$gt < gz \Longrightarrow t < z \Longrightarrow ft = gt = fz \Longrightarrow t = z \Longrightarrow fz = gt = gz,$$

contrary to the definition of z.

Corollary A.1. *If there is such an isomorphism $f : X \to I \subset Y$ then it is unique.*

This follows at once from the Proposition above.

Proposition A.4. *Suppose X, Y are well-ordered sets. Then either there exists an order-preserving injection*

$$f : X \to Y,$$

or there exists an order-preserving injection

$$f : Y \to X.$$

Proof ▶. Suppose there is an order-preserving injection

$$f_x : I(x) \to J$$

onto an initial segment J of Y for every element $x \in X$. If $J = Y$ for some x then we are done. Otherwise

$$J = I(y),$$

where (from the last Lemma) y is completely determined by x, say

$$y = f(x).$$

Then it follows easily that

$$f : X \to Y$$

is an order-preserving injection.

Suppose there is no such map f_x for some $x \in X$. Let $z \in X$ be the smallest such element. If $u < v < z$ then the corollary above shows that f_u is the restriction of f_v to $I(u)$. It follows that the maps f_u for $u < z$ 'fit together' to give an order-preserving injection

$$f : I(z) \to Y.$$

More precisely, if $x < z$ then (from the definition of z) there is an order-preserving isomorphism

$$f_x : I(x) \to I(y),$$

where $y \in Y$ is well-defined. We set $fx = y$ to define an order-preserving injection

$$f : I(z) \to Y$$

onto an initial segment of Y, contrary to the definition of z.

If X, Y are two well-ordered sets, we say that the ordinal of X is $\leq$ the ordinal of Y if there exists an order-preserving injection $f : X \to Y$. Ordinals are even further from our main theme than cardinals; but nevertheless, every young mathematician should be at least vaguely familiary with them.

We denote the ordinal of $\mathbb{N}$ with the usual ordering (which we have observed is a well-ordering) by ω:

$$\omega = \{0, 1, 2, \dots\}.$$

If X, Y are well-ordered sets then so is the disjoint union $X + Y$, taking the elements of X before those of Y. This allows us to add ordinals. For example

$$\omega + 1 = \{0, 1, 2, \dots, \omega\},$$

where we have added another element after the natural numbers. It is easy to see that

$$\omega + 1 \neq \omega :$$

the two ordered sets are not order-isomorphic, although both are enumerable. So different ordinals may correspond to the same cardinal. Of course this is not true for finite numbers; there is a one-one correspondence between finite cardinals and finite ordinals.

Note that addition of ordinals is not commutative, eg

$$1 + \omega = \omega,$$

since adding an extra element at the beginning of $\mathbb{N}$ does not alter its ordinality.

A.4.1 The Well Ordering Theorem

The Axiom of Choice states that for every set X we can find a map

$$c : 2^X \to X$$

such that

$$c(S) \in S$$

for every non-empty subset $S \subset X$.

We call such a map c a *choice function* for X.

The Well Ordering Theorem states that *every set X can be well-ordered.*

Proposition A.5. *The Axiom of Choice and the Well Ordering Theorem are equivalent.*

Proof ▶. If X is a well-ordered set then there is an obvious choice function, namely

$$c(S) = \mu(S).$$

The converse is more difficult.

Suppose c is a choice function for X. Let us say that a well-ordering of a subset $S \subset X$ has property $\mathcal{P}$ if

$$\mu(S \setminus I) = c(X \setminus I)$$

for every initial segment $I \subset S$ (except S itself).

We note that such a subset S must start with the elements $x_0, x_1, x_2, \ldots, x_\omega, \ldots$ unless S is exhausted earlier.

Lemma 8. ?? *A subset $S \subset X$ has at most one well-ordering with property $\mathcal{P}$.*

Proof ▶. Suppose there are two such well-orderings on S. Let us denote them by $<$ and $\subset$, respectively. If $x \in S$ let us write

$$I_<(x) \equiv I_\subset(x)$$

to mean that not only are these initial subsets the same but they also carry the same orderings.

If this is true of every $x \in S$ then the two orderings are the same, since

$$u < v \Longrightarrow u \in I_<(v) = I_\subset(v) \Longrightarrow u \subset v.$$

If however this is not true of all x, let z be the least such *according to the first ordering*:

$$z = \mu_< \left(\{ x \in X : I < (x) \not\equiv I_\subset(x) \} \right).$$

If $u, v \in I_<(z)$ then

$$u < v \Longrightarrow u \in I_<(v) = I_\subset(v) \Longrightarrow u \subset v.$$

It follows that $I_<(z)$ is also an initial segment in the second ordering, say

$$I_<(z) = I_\subset(t).$$

Hence

$$z = \mu_<(X \setminus I_<(z)) = c(X \setminus I_<(z)) = c(X \setminus I_\subset(z)) = \mu_< \left(X \setminus I_\subset(t) \right) = t.$$

Thus

$$I_<(z) = I_\subset(z).$$

Since, as we saw, the two orderings coincide, it follows that

$$I_<(z) \equiv I_\subset(z),$$

contrary to hypothesis. So there is no such z, and therefore the two orderings on S coincide.

Lemma 9. *Suppose the subsets $S, T \subset X$ both carry well-orderings with property $\mathcal{P}$. Then*

$$\text{either } S \subset T \text{ or } T \subset S.$$

Moreover, in the first case S is an initial segment of T, and in the second case T is an initial segment of S.

Proof ▶. As before, we denote the well-orderings on S and T by $<$ and $\subset$, respectively.

Consider the elements $x \in S$ such that the initial segment $I = I_<(x)$ in S is also an initial segment in T. By the last Lemma, the two orderings on I coincide.

If

$$I_<(x) = T$$

for some x we are done: T is an initial segment of S. Otherwise

$$I = I_<(x) = I_\subset(x),$$

with

$$x = c(X \setminus I).$$

Suppose this is true for all $x \in S$.

If S has a largest element s then

$$S = I_<(s) \cup \{s\} = I_\subset(s) \cup \{s\}.$$

Thus $S \subset T$, and either $S = T$,

$$S = I_\subset(s'),$$

where s' is the successor to s in T. In either case S is an initial segment of T.

If S does not have a largest element then

$$S = \cup_{x \in S} I_<(x) = \cup_{x \in S} I_\subset(x).$$

Thus S is again an initial segment of T; for

$$u \in S,\ v \in T,\ v \subset u \Longrightarrow v \in I_\subset(u) = I_<(u) \Longrightarrow v \in S.$$

Now suppose that $I_<(x)$ is *not* an initial segment of T for some $x \in S$. Let z be the smallest such element in S.

Since $I_<(z)$ is not an initial segment of T there is an element $t in T$ such that

$$t < z \quad z \subset t.$$

We are now in a position to well-order X. Let us denote by $\mathcal{S}$ the set of subsets $S \subset X$ which can be well-ordered with property $\mathcal{P}$; and let

$$U = \cup_{S \in \mathcal{S}} S.$$

We shall show that U is well-ordered with property $\mathcal{P}$.

Firstly we define a total ordering on U. Suppose $u, v \in U$. There exists a set $S \in \mathcal{S}$ containing u, v; for if $u \in S_1,\ v \in S_2$, where $S_1, S_2 \in \mathcal{S}$ then by Lemma 9 either $S_1 \subset S_2$, in which case $u, v \in S_2$ or $S_2 \subset S_1$, in which case $u, v \in S_1$.

Also if $u, v \in S$ and T then by the same Lemma the two orderings are the same. Thus we have defined the order in U unambiguously.

To see that this is a well-ordering, suppose A is a non-empty subset of U; and suppose $a \in A$. Then $a \in S$ for some $S \in \mathcal{S}$. Let

$$z = \mu(A \cap S).$$

We claim that z is the smallest element of A. For suppose $t < z,\ t \in A$. Then $t \in I(z) \subset S$, and so $t \in A \cap S$, contradicting the minimality of z.

Finally, to see that this well-ordering of U has property $\mathcal{P}$, suppose I is an initial segment of U, $I \neq U$. Let z be the smallest element in $U \setminus I$; and suppose $z \in S$, where $S \in \mathcal{S}$. Then $I = I(z)$ is an initial segment in S, with

$$z = \mu_S(S \setminus I) = c(X \setminus I),$$

since S has property $\mathcal{P}$.

Thus

$$U \in \mathcal{S}.$$

If $U \neq X$ let

$$u = c(X \setminus U),$$

and set

$$V = U \cup \{u\}.$$

We extend the order on U to V by making u the greatest element of V. It is a straightforward matter to very that V is well-connected, and has property $\mathcal{P}$. It follows that

$$V \in \mathcal{S} \Longrightarrow V \subset U,$$

which is absurd.

Hence $U = X$, and so X is well-connected.

Now we can prove the Comparability Theorem.

Theorem A.3. *Any 2 sets X, Y are comparable, ie*

$$\textit{either } \#(X) \leq \#(Y) \textit{ or } \#(Y) \leq \#(X).$$

Proof ▶. Let us well-order X and Y. Then by Proposition A.4 either there exists an injection

$$j : X \to Y,$$

or there exists an injection

$$j : Y \to X.$$

In other words,

$$\#(X) \leq \#(Y) \text{ or } \#(Y) \leq \#(X).$$