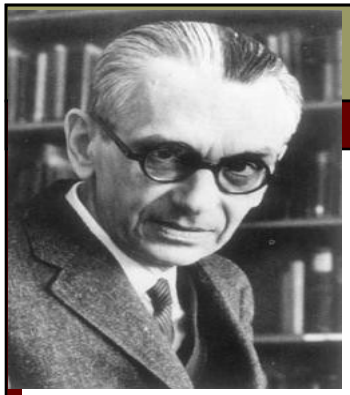
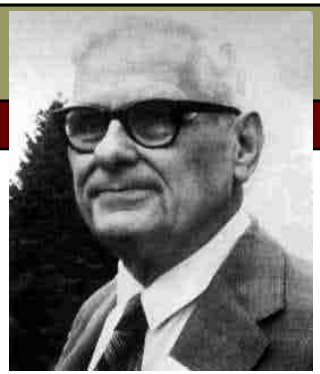
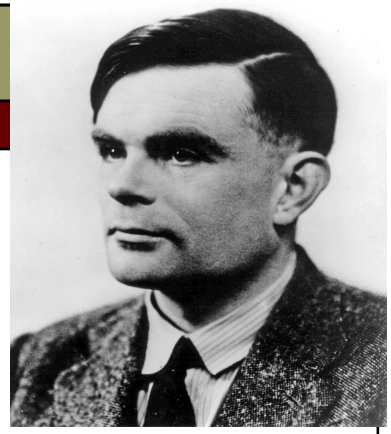
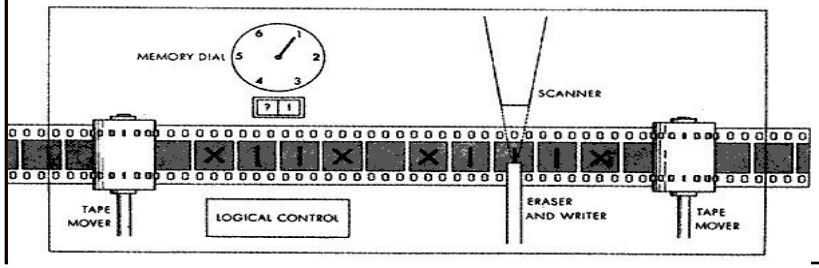


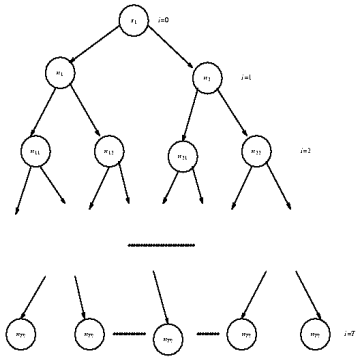




The theory of computability

The Turing degrees and relative computability





What is computability theory?

- In computability theory we study algorithms and computable functions from a precise mathematical perspective.
- We are not interested in the *best* way to solve a problem algorithmically – the efficiency of an algorithm is something we leave others to consider.
- We are interested quite simply in whether there exists any algorithm – in other words, any kind of computing device – which will compute a given function for you.
- In fact most functions turn out not to be computable. The bulk of mathematics is unknowable if one proceeds just like a machine.

Examples

- Examples of computable functions are easy to give. We are all happy, for example, that there exists an algorithm which will calculate addition on the natural numbers for us.
- An example of a non-computable function, is the function which takes Diophantine equations as inputs and outputs 0 or 1 according to whether the equation has a solution in the integers.

  A Diophantine equation is just an equation of the form

$$p(x_1, x_2, .., x_n) = 0$$

where p is a polynomial with integer coefficients.

.

Degrees of computability

- Since there are only countably many computable functions, most functions are non-computable. It becomes interesting, then, to consider ***degrees*** of computability – amongst the non-computable functions some are harder to compute than others…
- Given two functions f and g, they might be non-computable, but it might also be the case that IF you could compute f then you could compute g. If this is true and if it's also true that if you could compute g you could compute f, then these two functions are equally difficult to compute. We say they are of the same degree of computability.

Degrees of computability

- Now we can collect all functions together into degrees of computability.
- Let's order these degrees, so that one degree is higher than another if the functions inside the first degree are harder to compute than the functions inside the second. This gives us a structure which we call the Turing degrees. Much of computability theory is concerned with knowing what this structure looks like.
- Our formalization of the computable functions also allows us to formalize other notions of complexity. We can formalize the notion of random sequences and functions, for example, and then consider degrees of randomness for our various non-computable functions.

Hilbert's Programme

- Hilbert's programme aims at something like the complete mechanization of mathematics.
  - At the time the belief was that this programme would succeed – the belief was that there was nothing unknowable in mathematics.
  - In fact the programme would ultimately fail, but in order for this to occur it was necessary first to be able to give a formal mathematical definition of the computable (i.e. algorithmically calculable) functions.
  - In the 1920s and 1930s various formalizations were suggested, all were all proven to be equivalent.

The recursive functions

The partial recursive functions are the smallest set of functions U such that:

- The identity, constant and projection functions are in U;
- The functions in U are closed under composition, e.g. if h and g are in U then f(x)=h(g(x)) is in U;
- Closure under primitive recursion. If h and g are in U and f is defined:

$$f(x,0)=h(x)$$

$$f(x,y+1)= g(x,y,f(x,y))$$

  then f is in U;
- Closure under minimization. If g is in U and f(n) is the least y such that g(n,y)=0 if there exists such (and is undefined otherwise), then f is in U.

The Turing machine

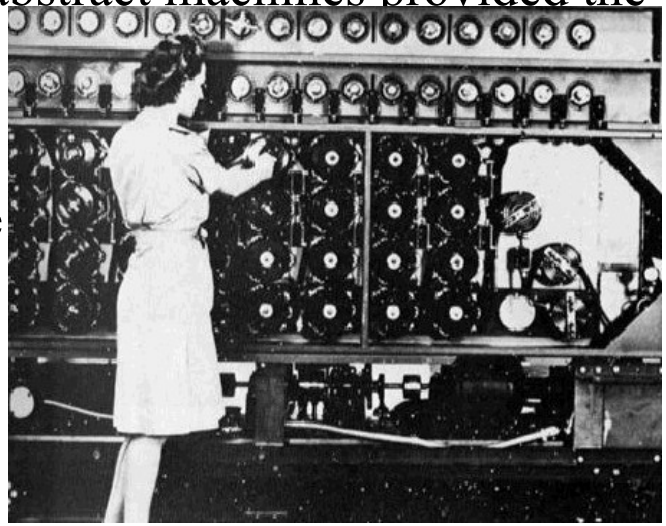
- It wasn't until Turing defined the class of Turing computable functions (in terms of machines that came to be called Turing machines) that people became convinced that we really had an appropriate formalization of the set of computable functions.

The Turing computable functions, the recursive functions (and all the other various formalizations) were proven to be equivalent.

The end of Hilbert's programme

- Since Godel had shown that, given any reasonably powerful language for discussing arithmetic, the set of true statements in the language is not a recursive set, and since we accept that a function is recursive iff there is an algorithm for calculating it, we conclude that there is no algorithm for deciding the true statements of arithmetic. This spells the end for Hilbert's programme.
- The timing here was interesting – Turing's abstract machines provided the conceptual basis for the modern day computer.

In world war II Turing helped to build the computing devices which were used to crack the enigma code.



The halting problem

- According to Turing's definition there are a countable number of Turing machines -- we can consider them to be listed so that there is one machine corresponding to each natural number.
- Upon being given some natural number input n, a Turing machine may never terminate and give an output, or it may terminate and give output some natural number m. We let

$$\phi_i(n)$$

denote the output of Turing machine number i given input n (so it may be the case that $\phi_i(n)$ is undefined).

The halting problem

- An important fact is that there exists a UNIVERSAL Turing machine – one that can simulate all others. Let <.,.> be a computable bijection $\omega \times \omega \mapsto \omega$. There exists i such that

$$\phi_i(\langle n, m \rangle) \simeq \phi_n(m)$$

• The task of deciding whether any given Turing machine terminates on any given input is not a computable task. This can be proved using a simple diagonal argument.

The halting problem

- Suppose that the halting problem is computable. Then the following function is computable:

$$f(n) = \begin{cases} \phi_n(n) + 1, & \text{if } \phi_n(n) \downarrow; \\ 0, & \text{otherwise.} \end{cases}$$

But if this function is computable it is ϕ_n for some n.
But f disagrees with ϕ_n on argument n.

Oracle Turing machines

Turing then went on to define oracle Turing machines – these are machines which have an extra input tape, called the oracle tape, on which we can write the values of any given function f. Once again there are countably many of these oracle machines, we write $\Psi_i(f, n)$ to denote the output of the ith oracle machine with oracle input f and on argument n. We say g is Turing reducible to f, denoted $g \leq_T f$ if there exists i such that $\Psi_i(f) = g$, i.e. $(\forall n)\Psi_i(f; n) = g(n)$.

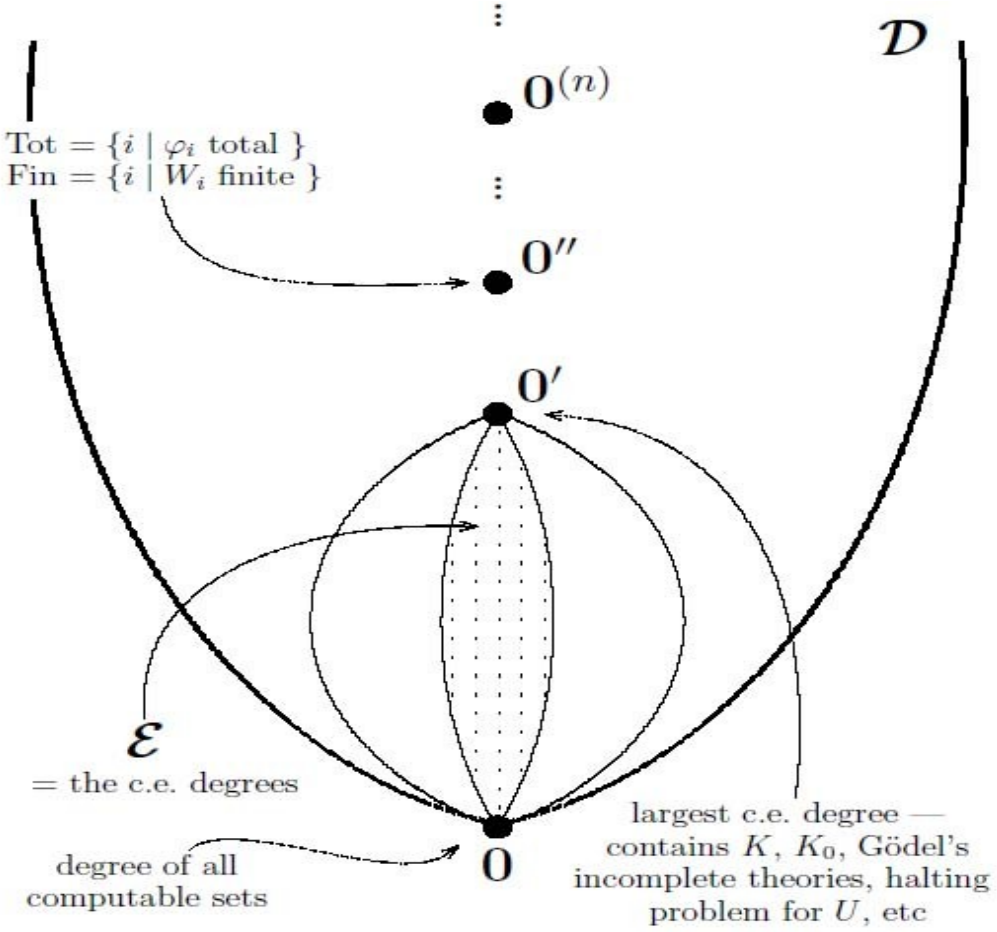
If $f \leq_T g$ and $g \leq_T f$ then we say that they are Turing equivalent, denoted $f \equiv_T g$. The Turing degree of f is the set of all functions which are Turing equivalent to it.

The Turing degrees

Now that we have defined the Turing degrees, there is an obvious way to put an ordering on them. For Turing degrees a and b, we say $a \leq b$ if $f \leq_T g$ whenever $f \in a$ and $g \in b$.

There is also an important function that we consider on this structure – the jump function is defined by relativizing the halting problem. If we fix f, then we can ask, for which n and m is it the case that $\Psi_n(f; m) \downarrow$? The jump function is the function which maps each degree to the degree of the halting problem of its members. This function is strictly increasing – so there doesn't exist a greatest degree.

The Turing degrees



The big questions

- Are there non-trivial automorphisms of the structure? How about for the local structures, such as the c.e. degrees?
- Can we find a simple definition of the jump function in terms of the ordering relation?
- There are many very basic structural questions which remain open…

Defining randomness

When presented with the two binary sequences

000000000000000000000000000000

011101001111100001010111111000

if asked to say which is most random…most people would say the second… but how to formalize this?

There are various approaches...and the nice thing is that they all turn out to be equivalent. The approach we shall consider here is that given by the notion of the compressibility of a sequence…

Defining randomness

The basic idea here is that the sequence of one million 0s is highly non-random…why?..because this is a long sequence which has a very short description.

In order to formalize this idea we need to formalize the notion of a short description. We fix some universal computer U and we say that the complexity of a word is the length of the shortest programme for U which produces the word. A sequence is random if its complexity is at least its own length – there is no programme for producing the sequence which is shorter than the sequence itself.

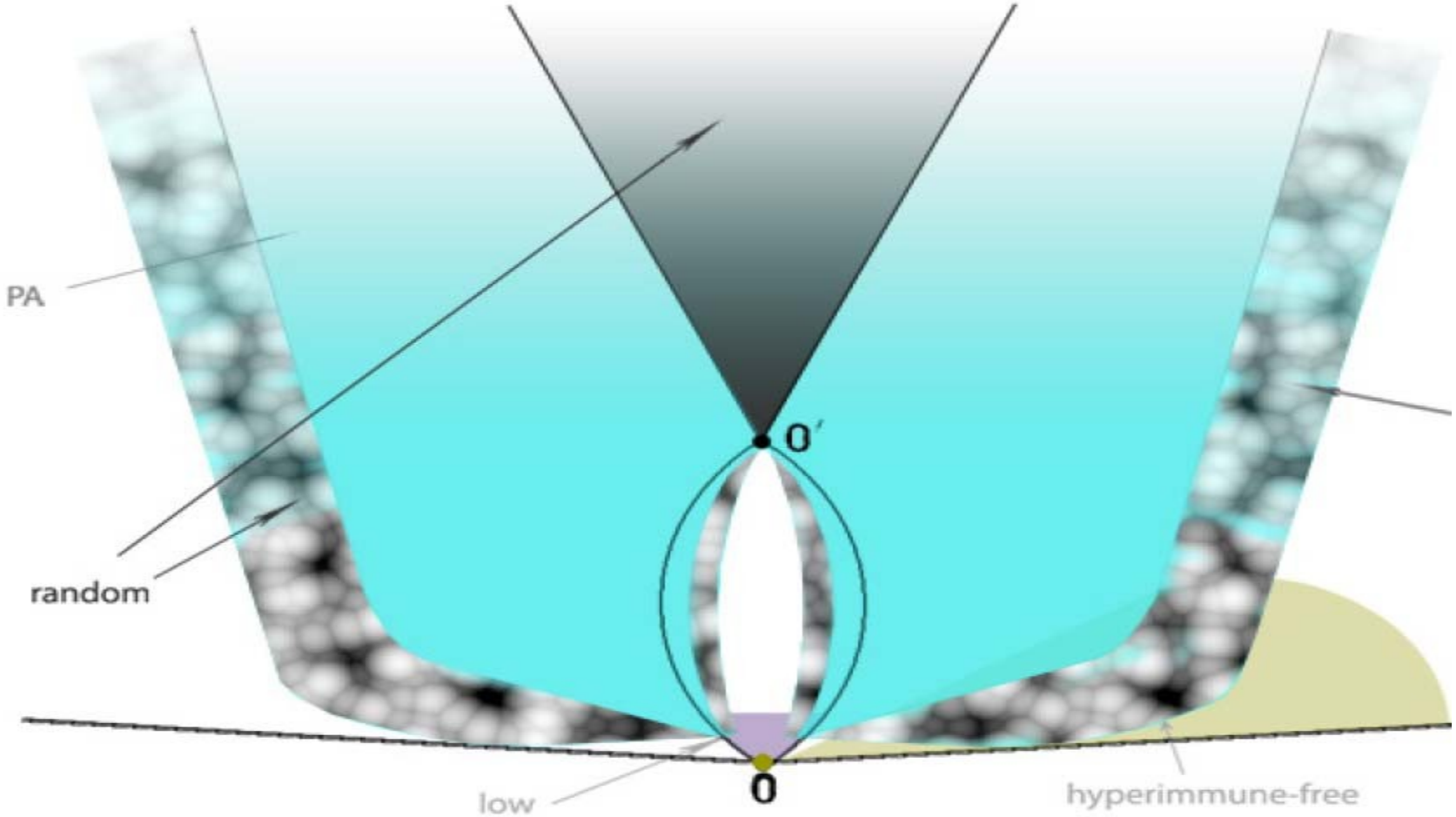
Extending to the infinite

- This gives us a notion of randomness for finite strings.. In order to extend the definition to infinite strings we just do the following.. We say an infinite sequence is random iff all its initial segments are random (there are some technical quibbles here as regards the fact that the domain of the machine should be prefix-free)…
- This definition satisfies all of the properties we would like. If we take any computable betting strategy, for example, and use it to guess the bits of an infinite random sequence then in the limit it will be correct half the time…

Random reals.

Now we have two fundamental measures of complexity – Turing complexity and randomness -- and so it seems a basic task to understand the relation between them. What are the Turing degrees of random reals? ?

Random degrees



Thanks for listening!