Infinite Sets & Computability.

Usually measure and compare set sizes by counting or calculating the number of elements they contain.

A set with 1708 members is bigger than a set with 920 members.

When sets are infinite, we can't put a number to their sizes, so can't compare sizes in the usual way. Deciding to say that the size of an infinite set is "∞" does not help. It just forces us to say that all infinite sets have the same size, because $\infty = \infty$. It prevents the real question from being asked

One to One Correspondence

If we can come up with a mapping, a direct-one-to-one correspondence between members of two sets, a mapping that misses nothing and never repeats anything, then those sets are obviously of the same size:

{red, green, blue, yellow} and {3, 7, 16, 92}

have the same size therefore/because I can come up with a correspondence, e.g.

red $\leftrightarrow$ 7, blue $\leftrightarrow$ 3, yellow $\leftrightarrow$ 92, green $\leftrightarrow$ 16

Unlike counting elements, this still works for infinite sets.

Natural Numbers & Even Numbers

$N = \{ 0, 1, 2, 3, 4, \ldots\ldots \}$

$E = \{ 0, 2, 4, 6, 8, \ldots\ldots \}$

I can make a $N \leftrightarrow E$ correspondence, not by listing all the N,E pairings, but by providing a procedure for finding the correspondence:

$a \in N,\ b \in E$: $a \leftrightarrow 2a$

or

$a \rightarrow 2a$

$\lfloor b/2 \rfloor \leftarrow b$

$\lfloor x \rfloor$ is the floor function: largest whole number $\leq x$.

every N maps to exactly one E
every E maps to exactly one N

Therefore

#Natural Numbers = #Even Numbers

similar reasoning brings in #Integers, etc

Strings are just Natural Numbers written in base 26 (or however big your set of letters is)

$"abc"_{26} = a \times 26^2 + b \times 26 + c$

perhaps a=0, b=1, c=2 etc

Strings are numbers written in a different base

#Strings = #Natural Numbers

Programs are just strings that pass certain tests.

#programs $\leq$ #strings.

The name for this infinite size is
aleph$_0$ $\aleph_0$

If it is possible to make a one-to-one correspondence between Natural Numbers and Things,

then #Things = $\aleph_0$

That correspondence means that an Enumeration can be made. A list of all possible things, that never repeats and will eventually reach any possible Thing.

```
for (NaturalNumber N=0; true; N+=1)
    cout << The_Thing_that_corresponds_with(N);
```

The ability to enumerate a set proves

#set = $\aleph_0$

If an enumeration can never possibly be produced then the opposite must apply.

#set ≠ $\aleph_0$

And if the set really is infinite, then

#set > $\aleph_0$

Proving that a particular list of Things is not an enumeration (i.e. it skips some or repeats some) means nothing.

Proving that no enumeration <u>can</u> exist means everything.

Think of the correspondence as a pair of functions; for example

when dealing with Natural Numbers & Even Numbers

$to : N \to E \qquad to(n) = 2n$

$from : E \to N \qquad from(e) = \lfloor e/2 \rfloor$

when dealing with Strings

$to : N \to String$

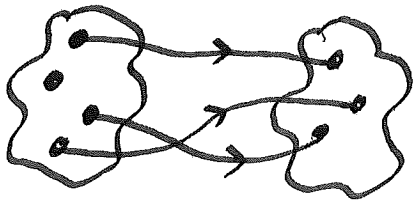
$from : String \to N$

The two functions to, from must be ~~bijections~~ total bijections

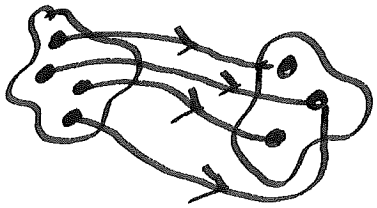
Bijection = Surjection ∧ Injection

Here is a picture of a function that is

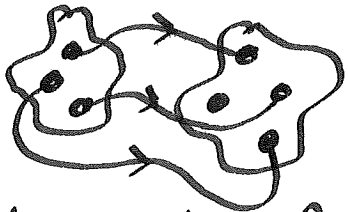
NOT total



NOT injective



NOT surjective



You never <u>have</u> to design the from function, as this is always sufficient:

If "to" exists, and is total, injective, and surjective this function is always correct.

```
Natural from (Thing x)
{ for (Natural n = 0; true; n+=1
    if (to(n) == x)
      return n; }
```

So, All of these sets have the same size

Natural numbers, Integers, Even numbers, Prime numbers, Strings, C++ programs, HTML files, ...

The whole lot. All have cardinality $\aleph_0$.

That isn't so amazing really. They are all the same kind of thing — strings of digits, strings of letters, all just lists of symbols.

TWO DIMENSIONAL INFINITY — CANTOR'S ENUMERATION

Consider the collection of all possible pairs of Natural Numbers:

(0,0) (0,1) (0,2) (0,3) (0,4), (0,5) (0,6)
(0,7) (1,0) (1,1) (1,2) (9,17) (6,2) (2,2)
(76394162789113, 100636094178821632) (3,1) etc

All those possibilities are infinite in two directions.

Until 1873 everyone (serious mathematicians included) had absolutely no idea of any of this. When Georg Cantor made this discovery it was rejected for publication because the academic establishment just refused to believe it.

A two-, three-, or million-dimensional infinity is no bigger than a one-dimensional infinity, but there really are bigger ones accessible by other means.

In 1873, Cantor discovered an enumeration for the set of all possible pairs of natural numbers.

This, in simple terms is the way to print his enumeration

```
for (NaturalNumber total=0 ; true; total+=1)
{ for (NaturalNumber first=0 ; first≤total ; first+=1)
  { NaturalNumber second = total-first;
    cout << "(" << first << "," << second << ")\n";}}
```

It produces the pairs in this order:

(0,0)
(0,1) (1,0)
(0,2) (1,1) (2,0)
(0,3) (1,2) (2,1) (3,0)
(0,4) (1,3) (2,2) (3,1) (4,0)
⋮

I hope it is easy to be convinced that this will never miss any possible (a,b) combination, and any combination you can think of will eventually be produced.

The "from" function of the enumerator is quite easy to derive:

```
NaturalNumber from(Pair p)
{ NaturalNumber a=p.first, b=p.second;
  NaturalNumber total=a+b;
  return total*(total+1)/2+a; }
```

The matching "to" function just requires a marginal facility with algebra. Try to work it out, remembering that from(a,b) = (a+b)*(a+b+1)/2 + a, and also remembering that 'to' is just the inverse of 'from'. Use the quadratic equation that we all remember from kindergarten, $x = \frac{1}{2a}(\sqrt{b^2-4ac} - b)$. If you need a clue, this is the solution

```
Pair to(NaturalNumber n)
{ NaturalNumber aplusb = floor(floor(sqrt(8*n+1)-1)/2);
  NaturalNumber zerob= aplusb*(aplusb+1)/2;
  NaturalNumber a= n-zerob;
  NaturalNumber b=aplusb - a;
  return Pair(a,b); }
```

Here, I used floor(x) to represent $\lfloor x \rfloor$, which is exactly what C++ does if you convert a float/double to an int using a typecast:

```
int floor(double x) {return (int) x;}
```

It is almost interesting to work out the to and from functions, and only requires pre-calculus mathematics. If you can't do it, please don't tell me.

Try out some examples:

from(3,1) ⇒ a=3; b=1; total=4; return 13

to(13) ⇒ aplusb = $\frac{\sqrt{8\times13+1}-1}{2}$ = 4;
zerob = 10; a=3; b=1;
return (3,1)

see! to = from^{-1}

This pair of to, from functions provides a one-to-one correspondence between Natural Numbers and pairs of Natural Numbers.

Let's be more technical:

$\mathbb{N} \equiv$ the set of all natural numbers

The <u>Cartesian Product</u> ("Cartes..." after René Descartes) of two sets A and B is written as $A \times B$, and means the set of all possible pairings of an element of A with an element of B.

e.g $\{a, b, c\} \times \{1, 2\} = \{ (a,1), (a,2), (b,1), (b,2), (c,1), (c,2) \}$

The cartesian product $\mathbb{N} \times \mathbb{N}$ is exactly the set that I've been going on about for three pages.

Thus, we have got a mapping (one-to-one) between $\mathbb{N}$ and $\mathbb{N} \times \mathbb{N}$. This proves that $\mathbb{N}$ and $\mathbb{N} \times \mathbb{N}$ are of exactly the same size.

Fractions, or Rational Numbers, the set $\mathbb{Q}$ are simply pairs of natural numbers with a different output format (instead of (a,b) print a/b) and a filter to prevent repetitions, such as 6/4 should not be printed if 3/2 already has been..

```
RationalNumber to(NaturalNumber n)
{ NaturalNumber found=0;
  for (NaturalNumber i=0; true; i+=1)
  { Pair p= Pair::to(i);
    if (gcd(p.first, p.second) != 1)
      continue;
    found+=1;
    if (found == n)
      return p.first / p.second; } }
```

This is exceptionally inefficient: To find the N^{th} Rational, search through all pairs $\in \mathbb{N}\times\mathbb{N}$ until you have found N that represent unique fractions, then stop.

But it does work

$$\#(\mathbb{N}\times\mathbb{N}) = \#\mathbb{Q} = \#\mathbb{N} = \aleph_0$$

If any two sets A,B have an enumerator (a "to" function) i.e. they have size $\aleph_0$, then the set of all combinations from A×B also has an enumerator, and therefore size $\aleph_0$:

```
PairofAB to(NaturalNumber n)
{ Pair p= Pair::to(n);
  return PairofAB( A::to(p.first) , B::to(p.second))
```

Do you get it?

if we are enumerating pairs of programs and strings:

to find the N^{th} such pair:

{ Find the N^{th} pair of NaturalNumbers = (a, b)

Return the a^{th} program paired with the b^{th} string

We already knew that

$\#Naturals = \aleph_0$

$\#Integers = \aleph_0$

$\#Strings = \aleph_0$

$\#Programs = \aleph_0$

Using Cantor's Enumeration, we therefore now know that

$\#Rationals = \aleph_0$

$\#PairsOfStrings = \aleph_0$

$\#CombinationsOfProgramsAndInputStrings = \aleph_0$

And because triples (a,b,c) are the same as pairs involving other pairs $(a,(b,c))$ we can extend this without limit:

$\#CombinationsOfThreeStrings = \aleph_0$

$\#PairsOfFractions = \aleph_0$

$\#CombinationsOfSeventyTwoNaturalNumbers = \aleph_0$

It is beginning to seem complete. Can we even imagine finding a set whose size $> \aleph_0$ after all these cases?

The INTEGER DECISION FUNCTIONS

these are functions that work out whether a given integer (or natural number to simplify it) have some property or not. Given a number, they test it and return 1 (for true) if the number has that property, or 0 (for false) if it hasn't.

Examples of Integer Decision Functions (simplified by expressing them in C++)

```
int is_even(int n)
{ if (n%2==0)
    return 1;
  else
    return 0; }
```

```
int is_twenty-four(int n)
{ if (n==24)
    return 1;
  else
    return 0; }
```

```
int is_prime(int n)
{ for (int x=2; x<n; x+=1)
    if (n%x==0)
      return 0;
  return 1; }
```

```
int is_less_than_17(int n)
{ if (n<17)
    return 1;
  else
    return 0; }
```

```
int is_perfect_square(int n)
{ for (int i=0; i*i<=n ; i+=1)
    if (i*i==n)
      return 1;
  return 0;}
```

A simple concept.

One way of representing an integer decision function would be as the (infinite) list of all of its results:

```
void print_representation( decision_function f)
{ for (int i=0; true; i+=1)
    cout << f(i); }
```

[if you care, the absurd C++ declaration is

```
void print_rep (int (*f)(int))
{ for (int i=0; true; i+=1)
    cout << f(i); }}
```

]

How many Integer Decision Functions are there?

They don't seem to be anything special, so let's assume $\#IDF = \aleph_0$, like everything else.

Given $\#IDF = \aleph_0$, we know it must be possible to produce an enumeration of I.D.F.s. There must be a function like

```
IntegerDecisionFunction to (NaturalNumber n)
{ return the Nth I.D.F.; }
```

[if you care, the even more absurd C++ declaration is

```
int (*)(int) to(NaturalNumber x)
{ .... }
```

or even worse; the prototype could be

```
int (*)(int) to (int);
```

]

Back to the point:

There is an enumerator: `IDF to(int n) {...}`

There is a representer:

```
void print(IDF f)
{ for(int i=0; true; i+=1)
    cout << f(i); }
```

With these two, we could represent an enumeration of IDFs as a two-dimensionally infinite table of 1s and 0s.

row \ column	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
3	0	0	1	1	0	1	0	1	0	0	0	1	0	1	0	0	0	1	0
4	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0
5	1	1	0	0	1	0	0	0	0	1	0	0	0	0	0	0	1	0	0
6	1	0	0	0	0	1	1	1	0	1	1	1	0	0	0	1	0	1	0
⋮	⋮	⋱																	

In this table, the digit at row R column C is the result that the R^{th} integer decision function would return if called with C as its parameter. In this example, IDF(0) is is_even; ~~IDF(1) is is_true~~ IDF(1) is always_false; IDF(2) is is_eleven; IDF(3) is is_prime; IDF(4) is is_perfect_square; IDF(5) is I don't know what.

In any enumeration of IDFs, every possible stream of 1's and 0's would appear in some row.

Given any table that looks at all like this, I can provide a stream of 1's and 0's (that is, an integer decision function) that is not in the table.

Let IDF(n) be the n^{th} integer decision function in some enumeration of IDFs.

f(x) is the result of applying function f to integer x

in particular

IDF(n)(x) is the result, 1 or 0, of applying the n^{th} integer decision function to the integer parameter x.
In the example table, IDF(3)(7)=1 because IDF(3)=is_prime, and 7 is prime

~B means "not B", ~0=1; ~1=0

I can always define this function:

$$Q(x) = \sim IDF(x)(x)$$

Q is clearly an integer decision function: for any int x it returns either 0 or 1, being the opposite of the entry at Row x, Column x in the table.

Does Q appear in the table?
The table is a list of all IDFs, so check them all:

is Q = IDF(0)? No, because Q(0) ≠ IDF(0)(0)
is Q = IDF(1)? No, because Q(1) ≠ IDF(1)(1)
is Q = IDF(2)? No, because Q(2) ≠ IDF(2)(2)

It goes on for ever.

$Q(x)$ is defined to be $\sim IDF(x)(x)$, or $\sim Table[x,x]$

so Q can not be the same as $IDF(n)$ ever

because $Q(n) = \sim IDF(n)(n)$

$\therefore Q(n) \neq IDF(n)(n)$

$\therefore Q \neq IDF(n)$

The significance of this is that we made just one assumption: IDF's can be enumerated, and produced from that assumption an impossibility. This means that the assumption was false.

Specifically:

Assume we've got an enumeration of IDFs

Find an IDF that can not be in that enumeration

An enumeration of IDFs must, by definition, include all IDFs

Therefore what we've got can't really be an enumeration.

The assumption contradicts itself.

REDUCTIO AD ABSURDUM

"reduction into an absurdity"

If an assumption leads to the impossible, then the assumption is false.

The assumption was that we've got an enumeration of IDF's

Therefore we can't have an enumeration of IDFs

So $\#IDF \neq \aleph_0$

but there are certainly an infinite number of them, therefore $\#IDF > \aleph_0$

There are more IDFs than anything else we've considered yet.

In particular, the number of integer decision functions is greater than the number of possible pairings of programs and inputs.

∴ No matter what programs you write, or what inputs they get while running, there still can't be enough of them to compute all of the integer decision functions.

i.e. There are some UNCOMPUTABLE INTEGER DECISION FUNCTIONS.

IDFs are the very basis of programming - the fundamental point of it - just answering yes/no questions about numbers.

Some (infinitely many) yes/no questions about numbers can never be answered.

There are some very simple uncomputable problems.

This is the essence of Turing's Thesis, the only significant fact in the theory of computation.

Well, maybe not "only significant", but certainly most significant.

EQUIVALENCES

Integer Decision Functions can be taken as descriptions of subsets.

The set of all prime numbers is simply the set of all Natural Numbers for which is_prime = 1

The 1's and 0's in the enumeration table are just selectors – a 1 means that the number at the top of this column is in the set.

The 0-1 sequence 00110101000101.....
describes the set {2, 3, 5, 7, 11, 13, ...

We know there are uncomputable IDFs, so now we know that there are uncomputable subsets.

But more interestingly, it gives us a numeric anchor.

How many subsets of {a, b, c} are there? Eight.
They are {}, {a}, {b}, {c}, {a, b}, {a, c}, {b, c}, {a, b, c}

always N items in a set $\Rightarrow 2^N$ possible subsets.

so, $\aleph_0$ elements in $\mathbb{N}$ means $2^{\aleph_0}$ subsets of $\mathbb{N}$ and $2^{\aleph_0}$ I.D.F.s

$2^{\aleph_0}$ really is $> \aleph_0$

$2^{\aleph_0}$ is called $\aleph_1$, and is known to be a genuinely bigger infinity.

There is another useful thing that unlimited sequences of zeros and ones represent. Just imagine that they are binary digits and that there is a "decimal" point to the left.

is_prime $\Rightarrow$ 0011010101000101.....

$\Rightarrow$ 0.0011010101000101..... base 2

$= 2^{-3} + 2^{-4} + 2^{-6} + 2^{-8} \rightarrow 2^{-12} + 2^{-14} - \ldots$

$\tilde{=}$ 0.20734.....

every IDF also represents a real number $\geq 0, < 1$

Integer Decision Functions
= # subsets of Natural Numbers
= # subsets of any $\aleph_0$ sized set
= # Real numbers between 0 and 1
= # Real numbers
= # Complex numbers
= # points in real 3-D space

remember that adding a dimension makes no difference.

Real Numbers > # program/input combinations

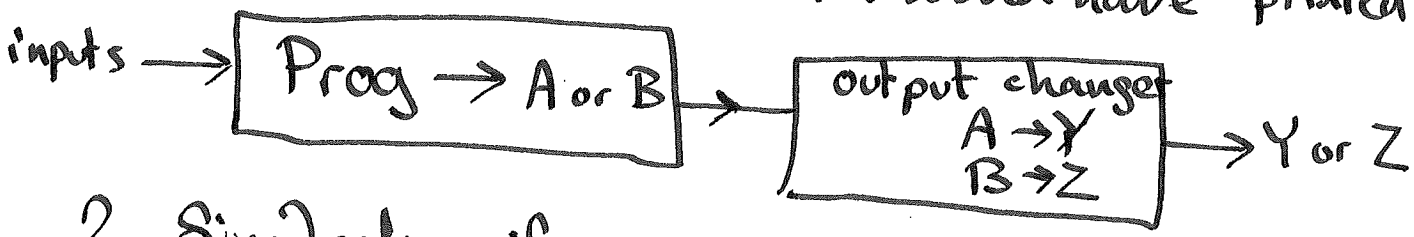
$\Rightarrow$ There are Uncomputable Numbers

There are things as simple as numbers that can never be known.
Computing Power has Serious Limits.

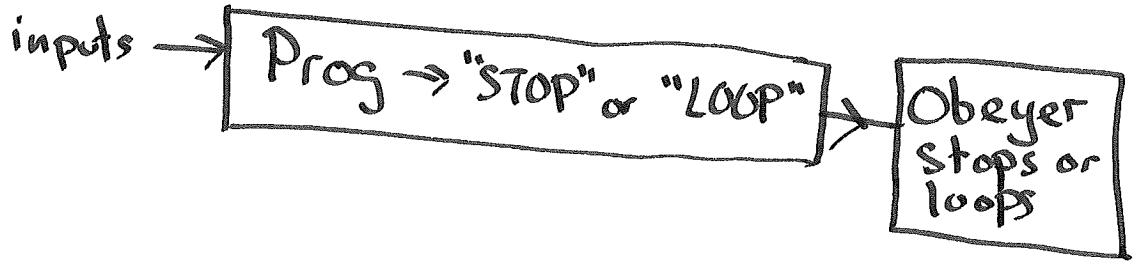
THE HALTING PROBLEM

Some axioms of programming:

1. If you can write a program that sometimes prints one string A, and sometimes prints a different string B,
   Then you can easily and always modify it to create an equivalent program with different outputs, printing Y when the original would have printed A, and Z when it would have printed B.

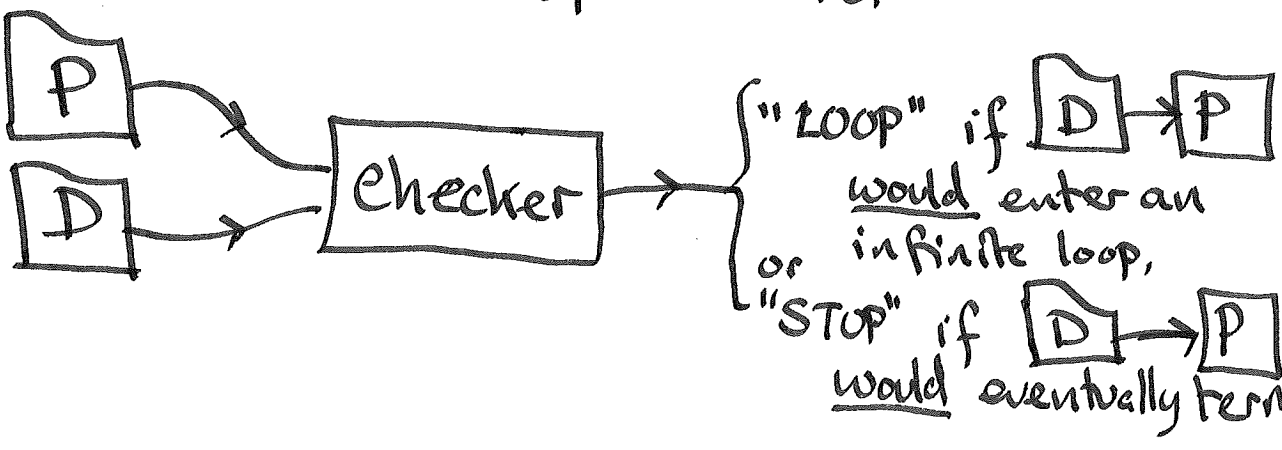


2. Similarly, if you can write a program that sometimes prints "STOP" and sometimes prints "LOOP",
   Then you can easily and always modify it to create an equivalent program that executes "exit(0);" instead of printing "STOP", and executes "while (true){}" instead of printing "LOOP"

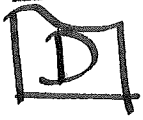


3. If you can write a program that opens and reads two different files in order to produce its results,
Then you can easily and always write an equivalent program that reads only one file, but uses it to produce both input streams.

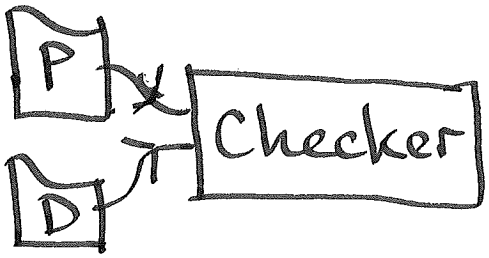
Now consider a special utility program. It checks programs to make sure they don't contain infinite loops. It does this by reading another program and the input file it will receive, and performing some very clever analysis. In fact, we don't care how it works, but it does. It analyses the input program to see what it would do. If the program would go into an infinite loop when given the data file, this checker utility prints "LOOP". Otherwise it prints "STOP"



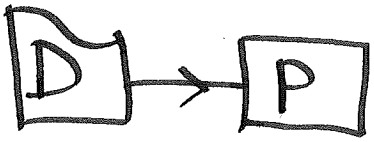
represents the program to be checked



represents a data file.

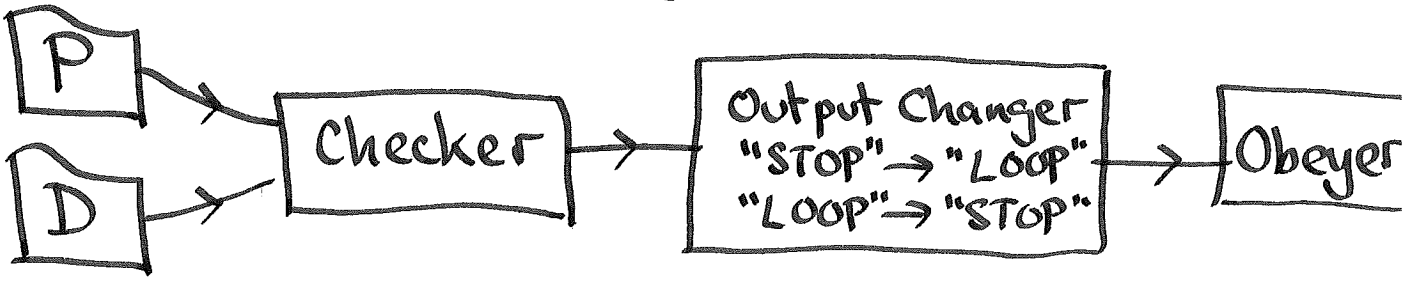


represents what would happen if the checker looks at P and D. It either prints "LOOP" or prints "STOP"



represents what would happen if program P were run, with D as its input. It may do a lot of things, but ultimately it will either come to a halt, or run for ever.

Now put these things together



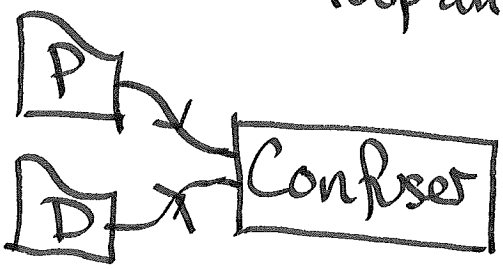
If when program P ran, reading D as its input, it would either run for ever or eventually stop.
The Checker finds out which, and prints "LOOP" or "STOP"
The Changer swaps these two possible outputs around, and the Obeyer actually does what that final result say.

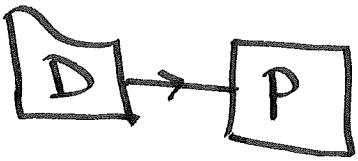
If P given D as input would run for ever, then this construction given P and D is input would stop.
If P given D would stop at some point, then this thing, given P and D as input would run for ever.

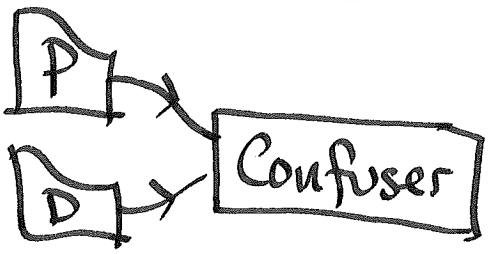
Let's give it a name:

is combined into one utility program:

So, if  would get into an infinite loop and never stop,

then  would eventually come to a stop and finish

But if  would eventually complete its task and stop running

then  would deliberately go into an infinite loop.

Or more concisely,

Confuser(P,D) = Obeyer(Changer(Checker(P,D)))

if P(D) ever terminates, Confuser(P,D) loops infinitely

if P(D) loops infinitely, Confuser(P,D) stops at some point.

Now using the pointless third axiom—
a single input stream can be forked and provided as both inputs to a two-input program:

this will also be given a name as a new program

≡ 

The important thing is that if we were given a Checker, then this Contradictor could clearly be created.

The input file F could be anything. If it is in fact a program, we get somewhere.

Contradictor(F) works out what would happen if the supposed program F were to read itself as input. If F(F) would run for ever, then Contradictor(F) would at some point stop. If F(F) would ever stop, then Contradictor(F) would deliberately run for ever.

Contradictor is itself a program. Why can't it be analysed?

What is the behaviour of Contradictor(Contradictor)

We just worked out exactly that:

If Contradictor(Contradictor) would run for ever
then Contradictor(Contradictor) would stop,
but if Contradictor(Contradictor) would ever stop
then Contradictor(Contradictor) would never stop.

Contradictor(Contradictor) must do the exact opposite of whatever it actually does.

Another reductio ad absurdum.
The only assumption was that we could have a checker. The assumption must be false, therefore we can not have a checker.

It is logically impossible to have a program that checks another program to see if it would ever halt.

This is an example of an integer decision function (programs ≅ strings ≅ natural numbers) → 1 (for "LOOP") or 0 (for "HALT").
A clearly understood, perfectly defined, useful function that is completely uncomputable.

It isn't just weird irrational numbers; these are sensible well-defined functions that just can't be computed.

[At this point, my big pen broke]

Many programs are simple enough that we can predict all possible behaviours. It isn't always impossible to know if a particular program will halt. It is impossible to have any method that could check all potential programs.

In very simple terms:

The Checker is a machine, you insert a program to be analysed through a slot in the top. It thinks for a while, then one of two lights will turn on:

This program would never stop if you ran it

If you ran this program, it would eventually stop

If you had such a machine, it could work sometimes, but there will be an infinite number of programs for which it would not give a correct answer. In these cases, it would either give a wrong answer, or it would fail to produce any answer at all.

That is the Halting Problem

Imagine that you had a much simpler utility machine: You give it a program and point to one particular line or statement, and ask "will this particular statement be executed?" Yes or No.

If you had that machine, you could modify any program you like, so that every possible termination point — calls to exit(), returns from main(), etc — are replaced by function calls f(). Define the function f()={int x=1;}. Ask your new machine if that "int x=1;" statement will be executed. If it says yes, you know that the original program would have halted; if it says no, you know that it would have run forever.
This new machine would enable you to solve the halting problem.

Therefore, this new machine can not exist either.
All sorts of interesting problems can be demonstrated to be uncomputable by showing that a solution to them would provide a solution to the halting problem, a known impossibility.

A Philosophical Electronic Escape?

Computers are deterministic things: what they do next is entirely determined by their state — all the bits of memory, registers, switches, flip-flops and so on.
There are a large number of such state bits in a computer, especially if you take disc storage into account. Let's name it: N = number of state-determining bits in computer C.

Now, with N bits, there are at most 2^N possible combinations, or states that the computer could be in. If a computer ever returns to a state <u>exactly identical</u> to one it has been in before, it must repeat its previous action and thereby enter an infinite loop.

So what if we monitored all those bits: after every operation, we look at them and see if we have ever been in that state before. If we have, then there's an infinite loop. As there are only 2^N different states, then by the time 2^N operations have been executed we must either have repeated a state, or stopped.

All we need to do is keep track of which states a computer has been in before (just one single bit of information for each possible state) and the Halting Problem is solved.

The problem is that if a computer has N bits of state (memory etc) then 2^N bits would be needed to keep this "have we seen this state before" information.

and 2^N is always $> N$

So a computer would need to have more memory than it has got (however much that is) to use this solution. Logically Impossible.

If you had a really tiny computer, say with only 1K bytes of memory, A really huge computer, with approximately 10^{2500} bits of memory, could monitor the tiny computer and solve the Halting Problem for it.

A vastly superior being can fully analyse, and know the entire "mental state" of a really humiliatingly puny and inferior being, and therefore answer the Halting Problem for it. But that is not useful. The superior computer could not be checked. A real checker always needs more memory than it could possibly have.

Real Philosophy

What about an infinitely superior being, with infinite memory available to it?
Unfortunately for him, it wouldn't help. 2^x is the one thing <u>always</u> guaranteed to be strictly greater than x, even for infinite numbers. Remember? $2^{\aleph_0} = \aleph_1$
So an infinite computer whose name might begin with G could completely analyse all the little mortal computers, but could never understand Himself. Omniscience is impossible even in the realms of the infinite.

Some people, possibly those with religions, might be unhappy with that last statement.

You are not going to be tested on it.