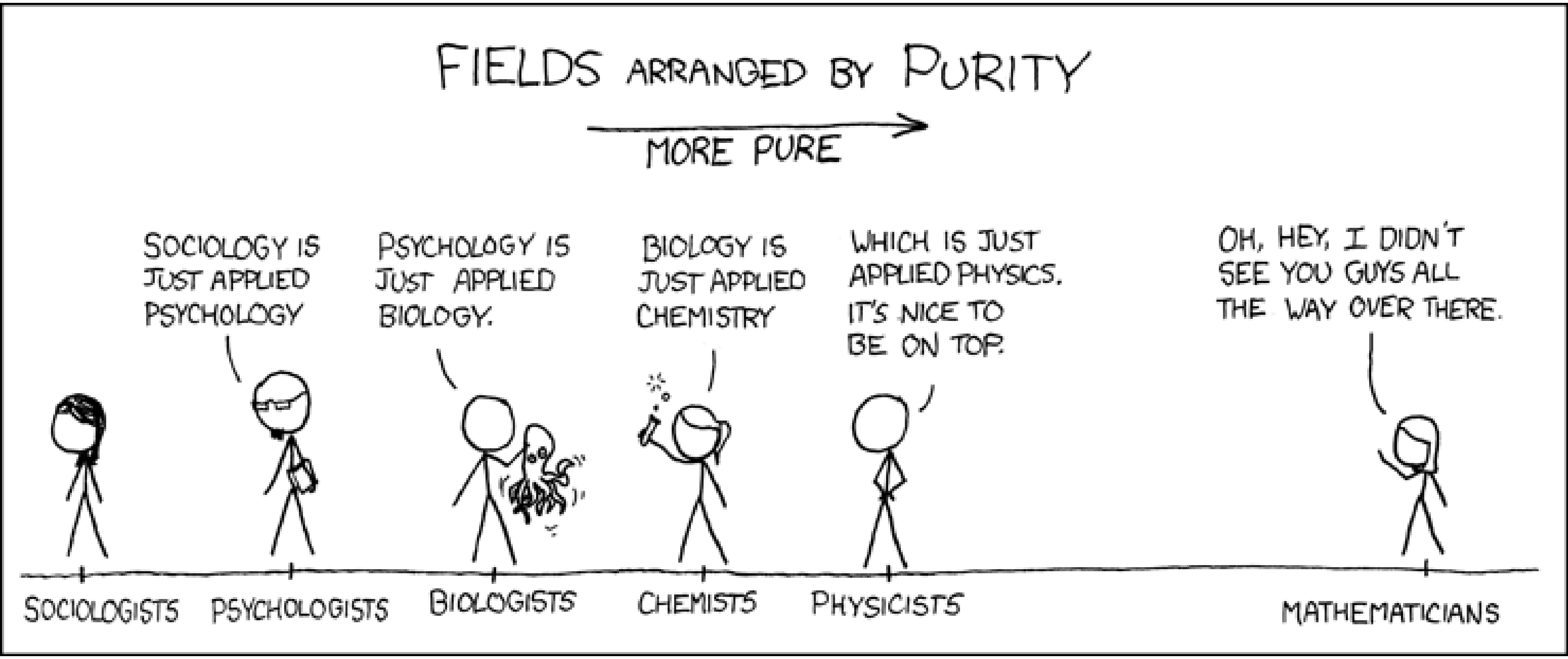
Infinity
and
decidability


FIELDS ARRANGED BY PURITY
MORE PURE
SOCIOLOGY IS JUST APPLIED PSYCHOLOGY
PSYCHOLOGY IS JUST APPLIED BIOLOGY.
BIOLOGY IS JUST APPLIED CHEMISTRY
WHICH IS JUST APPLIED PHYSICS. IT'S NICE TO BE ON TOP.
OH, HEY, I DIDN'T SEE YOU GUYS ALL THE WAY OVER THERE.
SOCIOLOGISTS
PSYCHOLOGISTS
BIOLOGISTS
CHEMISTS
PHYSICISTS
MATHEMATICIANS

Assumed properties of algorithms

- Certainly:

Deterministic: Given the same input, produces the same output.
Finite: It can be described in a finite number of steps
Definite: Each step has a clearly defined meaning.

- Nice to have:

Correct: It produces correct answers
Time Bounded: It eventually stops
Fast: it not only stops, but stops quickly

Decidability

- By a “**problem”** we will mean a precisely defined problem with a clearly understood set of potential answers. Mathematical problems are, in that sense, ok. “Problems” involving random input are not ok. “Where to dig to find a hidden treasure” is not ok. “How to be happy” is not ok. “Which number to bet on for roulette” is not ok.

- **Decidable problem**: there is an algorithm which tells a correct answer for each input or case of the problem.

- Examples of decidable problems: Is a number even? Is it prime? Does a quadratic equation have a solution? Does it have an integer solution? What is a solution of a quadratic equation which does have a solution? What is greatest common divisor of any two integers? Is there a winning move for a chess exercise (and If yes, which one)? Etc.

- **Not all (precise) problems are decidable**. In some sense one could say that the likelihood that a “random” problem is decidable, is infinitely small.

Intuitive explanation for undecidability

- There are infinitely more problems than algorithms.
- Hence there are not “enough” algorithms for problems.
- How we are going to prove this:
  - Easy: show that there are as many algorithms as there are integers.
  - Almost easy: show that there are at least as many precise problems as there are real numbers.
  - **Cantor’s theorem**: there are infinitely more real numbers than there are integers.

As many algorithms as integers

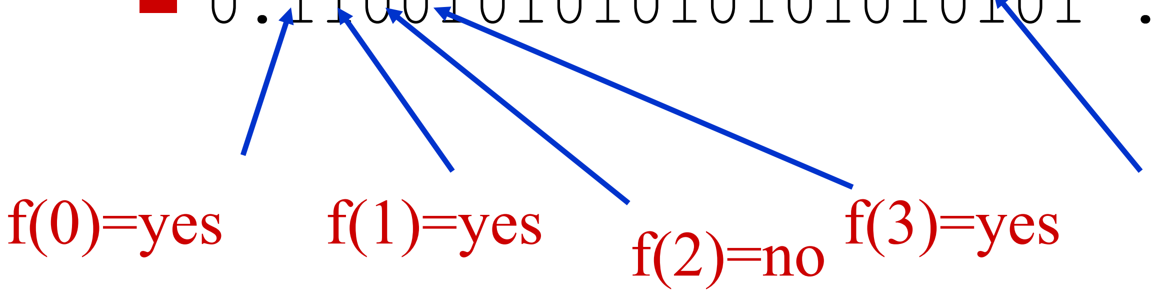
- Any algorithm can be written in any turing-complete programming language. Let us choose Python, for example.
- **Every Python program can be written in a single text file in ascii.**
- Ascii strings consist of bytes, each in the range 0-255.
- Each ascii string corresponds to an integer: treat is a number in the system of 256.
- like this:
  - Bytes 0 22 64 give a number 22*256+64
  - Bytes 64 120 68 give a number 64*256*256 + 120*256 + 64
- NB! Obviously, every ascii string is not a correct Python program.

As many problems as real numbers

- **A real number is a number which may have arbitrarily many decimal places.**
- Like: 2,232425453441231231...
- 2, pi, ¾, square root of two are all real numbers
- Each real number in binary contains digits 0 or 1, like: 110.1101001011010111101010101....

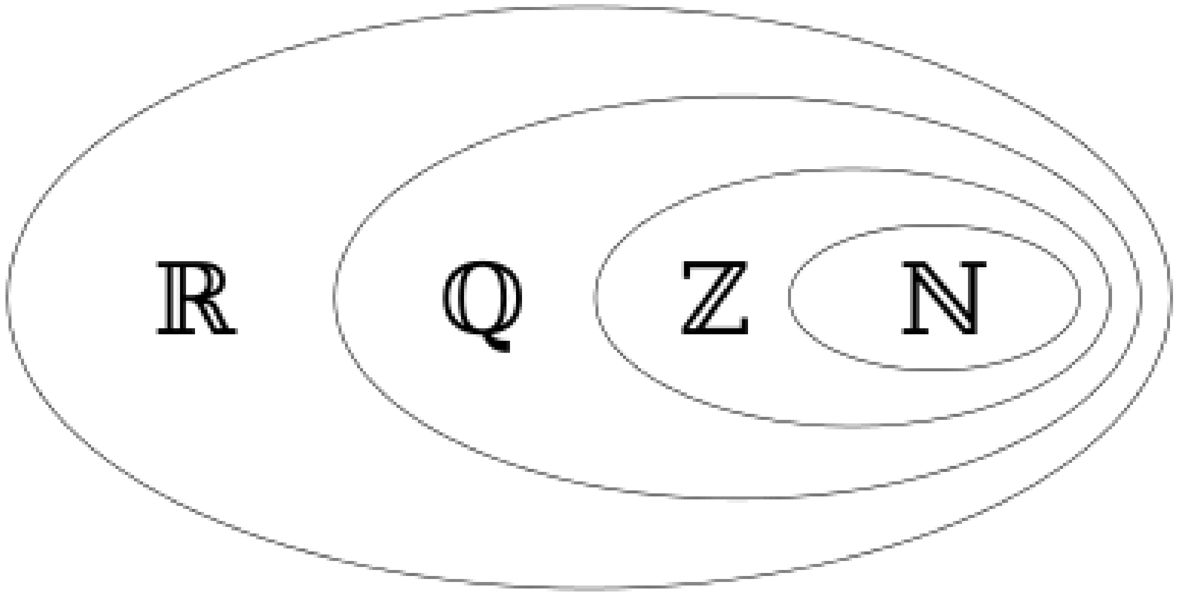
- Consider a specific class of problems: yes/no problems on integers. Like “is this number prime?”
- Algorithm takes an integer as input and must answer yes or no.

- How many such problems there are?
- Every binary real number under 1 corresponds to one such problem
- 0.110010101010101010101 ...



Generally:
f(n)= bit at place n

Real, rational, integers, natural numbers



Intro to the Cantor's theorem

- **The cardinality of the set of real numbers is bigger than the cardinality of the set of integers. In other words, there are more real numbers than integers.**
- Cardinality means roughly "size". For finite sets it means exactly "size".
- How about positive/negative integers and rational numbers?.
- Are there more neg/pos integers than positive integers?

```
...  -4    -3   -2    -1    0    1    2    3    4  ...
                            0    1    2    3    4  ...
```

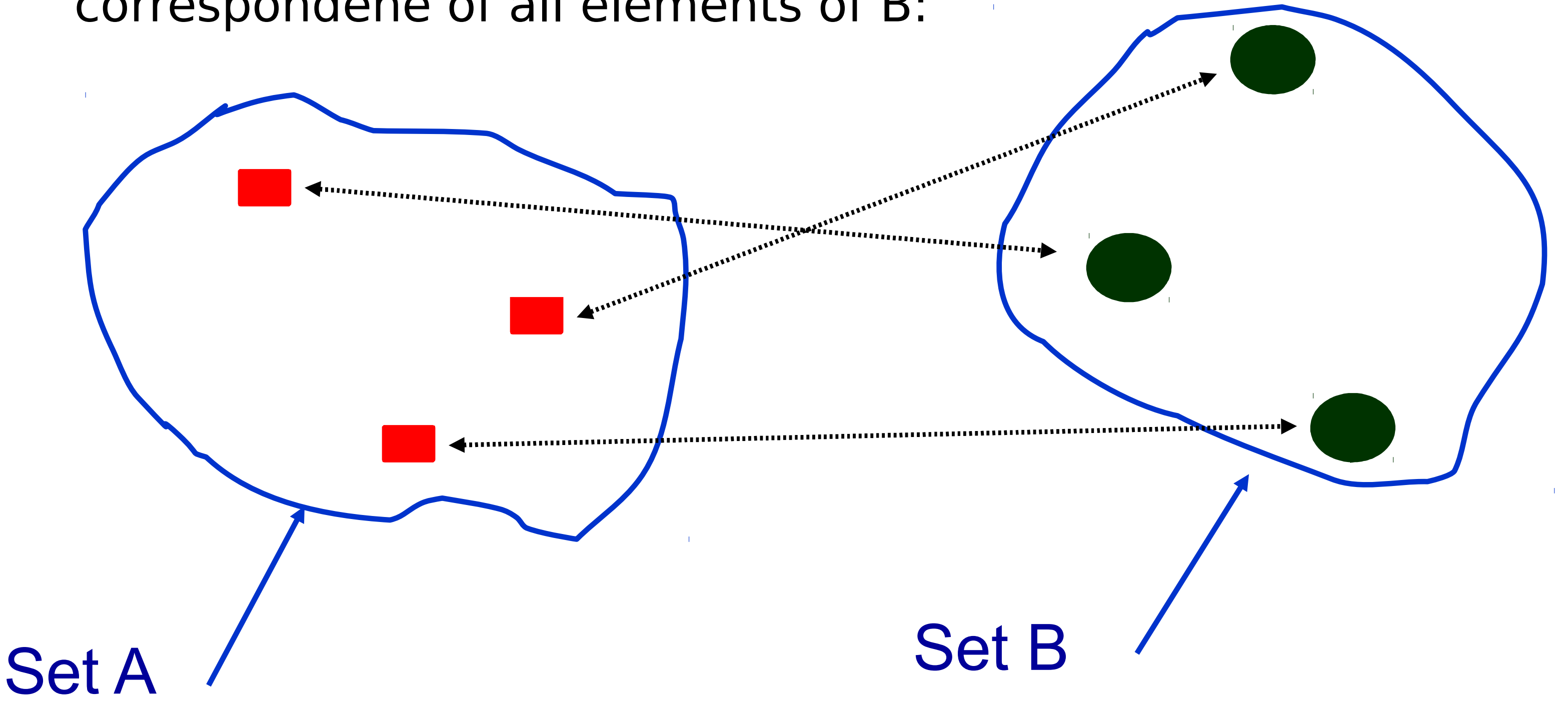
Neg/Pos integers: Z

Positive integers: N

Intro to the Cantor’s theorem

- What does it mean:
- “set A has the same cardinality as the set B”?

- Means: all elements of A can be set to one-to-one correspondene of all elements of B:



Each element of A corresponds to a single element of B and vice versa

Intro to the Cantor’s theorem

- One-to-one correspondence between sets Z and N?

```
Z:   ... -4   -3  -2   -1   0    1   2   3   4 ...
N:                          0    1   2   3   4 ...
```

- Z seems bigger than N?

Intro to the Cantor's theorem

- Instead of this:

Z:	...	-4	-3	-2	-1	0	1	2	3	4	...
N:						0	1	2	3	4	...

- Let us create a correspondence like this:

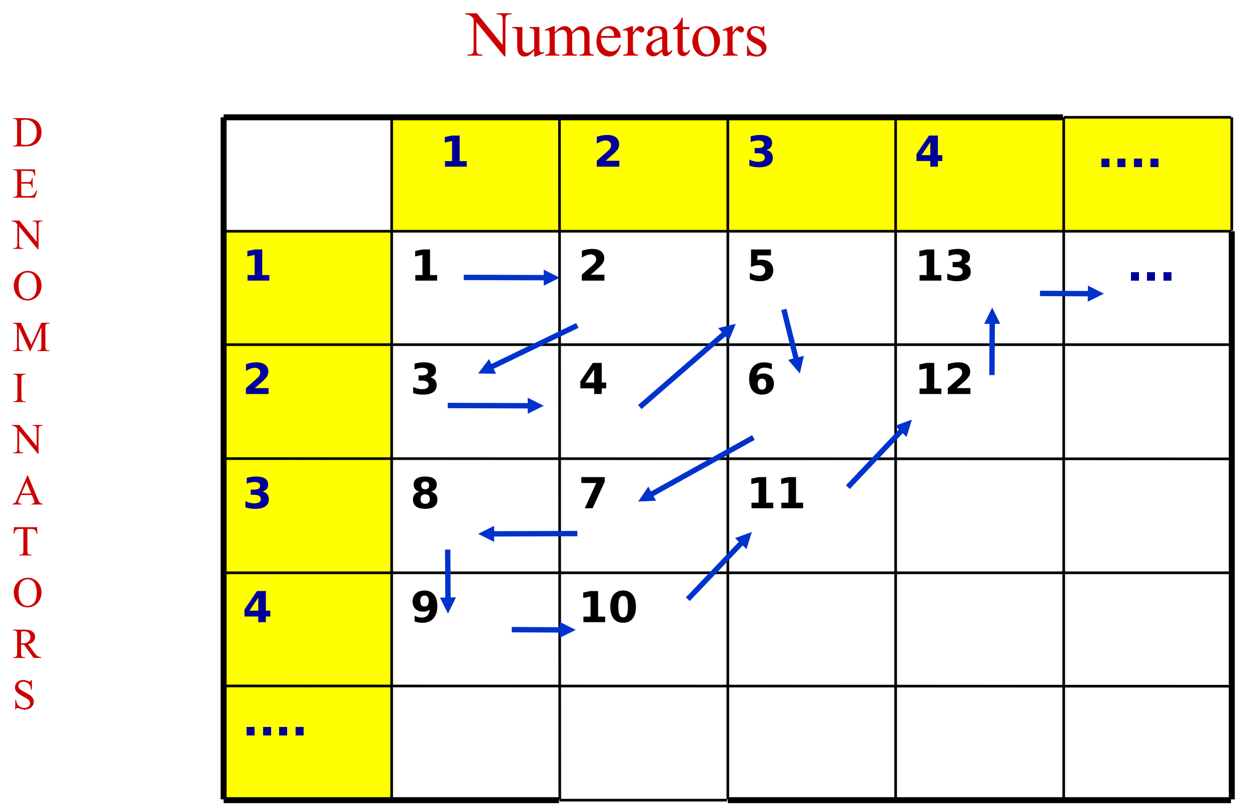
Z:	0	1	-1	2	-2	3	-3	4	-4	…
N:	0	1	2	3	4	5	6	7	8	…

Hence N and Z have the same cardinality!

Intro to the Cantor’s theorem

- How about rational numbers Q? There are infinitely many rational numbers between each pair of rational numbers.

One-to-one correspondence is possible:


Numerators
DENOMINATORS

	1	**2**	**3**	**4**	**....**
1	1	2	5	13	**...**
2	3	4	6	12	
3	8	7	11		
4	9	10			
....					

Cantor’s theorem: idea

- Create a table of all real numbers under 1:

Digits after the initial comma

NUMBERS

1	2	1	5	6	...
3	3	1	0	9	...
2	8	3	5	6	...
3	5	6	9	0	...
				...	

- Look at a diagonal **1 3 3 9 ...** and construct a new number, adding 1 to each digit of a diagonal (let 10 become 0) giving us **2 4 4 0 ...**

Cantor’s theorem

1	2	1	5	6	...
3	3	1	0	9	...
2	8	3	5	6	...
3	5	6	9	0	...
				...	

- First digit of a new number is 1+1 = 2.
- Our constructed new number: 0 . 2 ...

Cantor’s theorem

1	2	1	5	6	...
3	3	1	0	9	...
2	8	3	5	6	...
3	5	6	9	0	...
				...	

- **Second digit of a new number is 3+1=4**
- **Our new number: 0 . 2 4 ...**

Cantor’s theorem

1	**2**	**1**	**5**	**6**	**…**
3	**3**	**1**	**0**	**9**	**…**
2	**8**	**3**	**5**	**6**	**…**
3	**5**	**6**	**9**	**0**	**…**
				…	

- Our new number is **0 . 2 4 4 0 …**
- Is our new number on some line in the table?
- **Cannot be on the first row:**

  First digit of first row is **1**, first digit of our new number is **2**

Cantor’s theorem

1	2	1	5	6	...
3	3	1	0	9	...
2	8	3	5	6	...
3	5	6	9	0	...
				...	

- Our new number is **0 . 2 4 4 0 ...**
- Is our new number on some line in the table?
- **Cannot be on the second row:**
- Second digit of second row is 3, second digit of our new number is 4

Cantor’s theorem

1	2	1	5	6	...
3	3	1	0	9	...
2	8	3	5	6	...
3	5	6	9	0	...
				...	

- Not a random coincidence: we constructed the number so that
- **It cannot be on row N:**
  On row N digit N we had X, but the N digit of the new number is X+1 (if X= 9, then 0)

Cantor’s theorem summary

- Our new number 0. 2 4 4 0 …. cannot be in this table

- Had we created the table in a different order, then, after creating a new number from the diagonal, the new number would still not be in the table.

- Our new number is clearly a real number
- Hence no table of real numbers under 1 can contain all the real numbers under 1!

- Hence:
  - You cannot create a table of all real numbers
  - Each row N in the table corresponds to integer N
  - **Hence the set of real numbers has a bigger cardinality than the set of integers**

Continuation

Observe that:

- Each real binary number between 0 and 1 corresponds to one subset of integers. Value of bit N tells whether integer N is in that set or not.

- **Examples:**
  - Even integers: `101010101010101010....`
  - All integers: `11111111111111111....`
  - Primes: `0101010100 0101....`

- Cantor’s theorem says more generally “**the set of all subsets of any set H has a higher cardinality than H**”

- Hence there are infinite chains of larger and larger infinites.

Major open problem

Continuum hypothesis: are there sets with a cardinality between that of integers and that of reals?

Nobody knows.

We do know, though, that neither this hypothesis nor its negation can be derived from a large axiomatization of set theory called ZFC.

Halting Problem

- Some programs halt, some not:

```
def iamhalting(i):  # halts for any i
   while i<10:
      print i*i
      i=i+1

def iamnothalting(i): # does not halt if i<=5
   while i<10:
      print i*i
      if i>5: i=i+1
```

- You plan to write a program that will take in a user's program and inputs and decides whether
  - it will eventually stop, or
  - it will run infinitely in some infinite loop.

Some ideas for writing a halting checker

- Let the checker interpret a given program row-by-row: if the program eventually halts, the checker will detect that.

- How to check that the program **does not halt**?

  - Look for loops such as *while (statement)*
    - If variables in the statements are unchanged e.g.
      - *While (t<1)* with *t* is initialized to *0* but never used in the loop
      - No statements to get out of the loop, such as *goto* or *break*
    - then program will not halt.
  - Add additional checks and methods, each covering some cases and being able to detect non-halting for some programs.

- Can we detect all the reasons why the program does not halt?

Interesting case of a halting problem: 3n+1

```
# Collatz conjecture: seq3np1(i) seems to halt for all i.
# All tries for 1,2,....,huge_number have led to halting.
# But, nobody has found a proof that seq3np1(i) halts for all i.

# Print the 3n+1 sequence from n, halting when it reaches 1.

def seq3np1(n):
  while n != 1:
    print(n)
    if n % 2 == 0:    # n is even
      n = n / 2
    else:             # n is odd
      n = (n * 3) + 1
  print(n)            # the last print is 1
```

Interesting case of a halting problem: 3n+1

```
# It is easy to see that search() will never halt,
# regardless of whether seq3np1(i) halts for all i or not


def search():
  i=1
  while True:
    print "*** checking ",i
    # if seq3np1 does not stop for some i,
    # it will loop forever and never halt
    seq3np1(i)
    i=i+1
    # if the seq3np1 does stop for all i,
    # search() will still not halt
```

Interesting case of a halting problem: 3n+1

```
# Suppose we have a halting solver: Halts(function,param)
# It is easy to see that supersearch will halt if and only if
# seq3np1 does not halt on some i

def supersearch(dummy): # dummy is not actually used
  i=1
  while True:
    print "*** checking ",i
    # regardless whether seq3np1 does or does not stop,
    # the next line will halt:
    if Halts(seq3np1,i):
      i=i+1 # keep searching for non-halting i
    else:
      print "seq3np1 does not halt on",i
      return i
```

Interesting case of a halting problem: 3n+1

```
# If supersearch can be written and works ok, we can compute
# whether seq3np1 halts for all integers, thus solving a major
# open problem

if Halts(supersearch,1): # 1 is just a dummy value
  print "seq3np1 always halts"
else:
  print "seq3np1 does not halt on value"
  print Halts(supersearch,dummy)
```

Proof that a halting checker is impossible: start

- Assume that it is possible to write a program to solve the Halting Problem.
- Denote this program by **HaltAnswerer*(prog,inputs)*.**

```
HaltAnswerer(prog,inputs):
        if  halts (prog(inputs)):
         return true
  else:
         return false
```

- A program is just a string of characters
  - E.g. your Python program is just a long string of characters
- An input can also be considered as just a string of characters
- So HaltAnswerer is effectively just working on two strings

Proof part 2:

- We can now write another program **Nasty(prog)** that uses **HaltAnswerer** as a subroutine. The program **Nasty(prog)** does the following:

```
Nasty(prog):
    if HaltAnswerer(prog,prog)==true:
        # case 1: Nasty will go into an infinite loop
        while true: x=1
    else:
        # case 2: Nasty will halt
        return
```

- Consider what happens when we run **Nasty(Nasty)**.
- If **Nasty loops infinitely**,
  - HaltAnswerer(Nasty,Nasty) returns false which by [case 2] above means Nasty will halt.
- If **Nasty halts**,
  - HaltAnswerer(Nasty,Nasty) will return true which by [case 1] above means Nasty will loop infinitely.

- Conclusion: Our assumption that it is possible to write a program to solve the halting problem has resulted in a contradiction.

Diagonalization for halting problem: try it out!!

- Each program can be represented by a string and each string can be represented by a natural number

- **Create a table for all programs with a single input where the entry *(i,j)* is**
  - *H* if program *i* halts when program *j* is used as input
  - *NH* if program *i* does not halt when program *j* is used as input

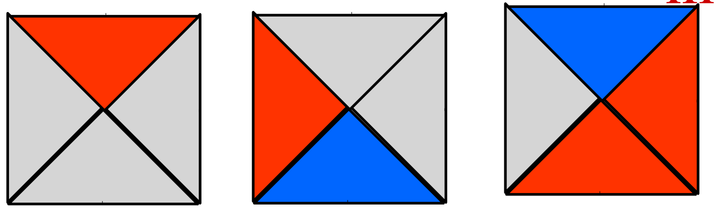
- **Where is *Nasty* in this list?**

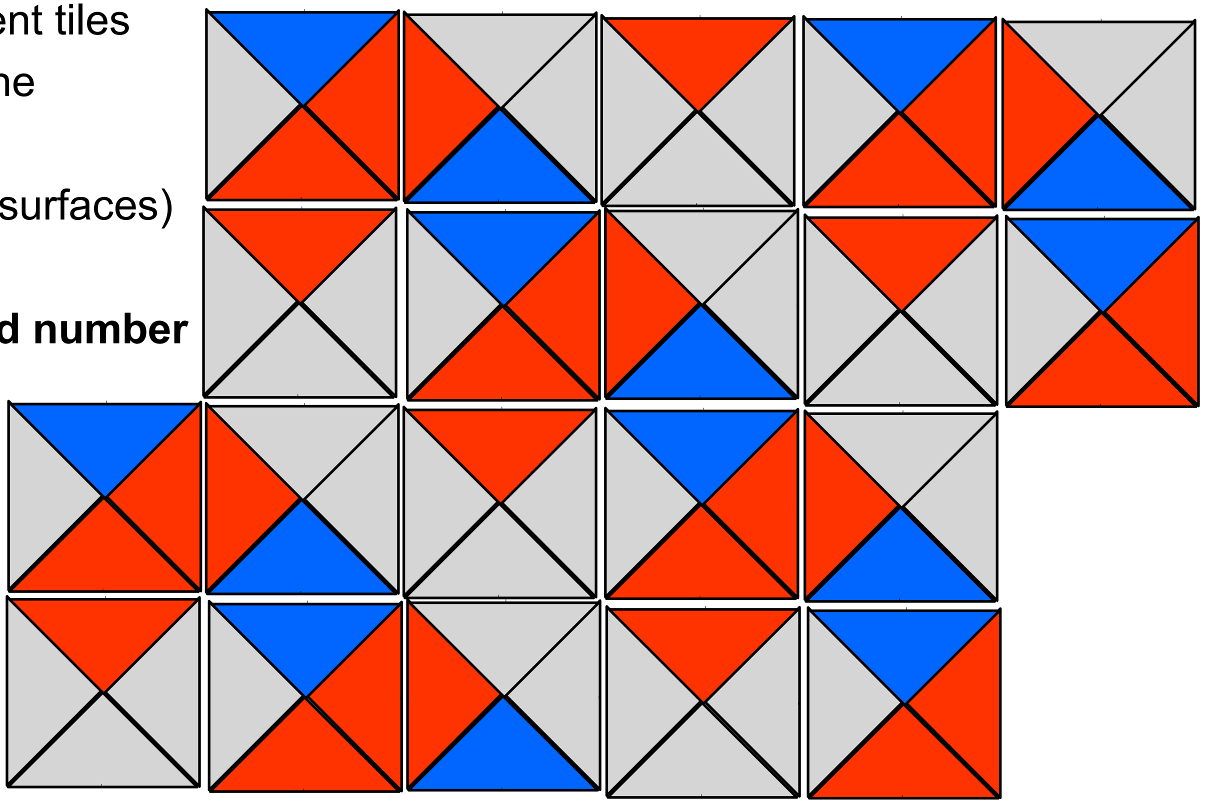
Example from geometry: “tiling problem”

Assume that you have
a finite set of tiles
(which cannot be rotated)
and you would like
to know if you can tile
an area using those
tiles (adjacent tiles
must have the
same color
on adjoining surfaces)

An **unlimited number of tiles** is available of each kind

Three tiles given in this example

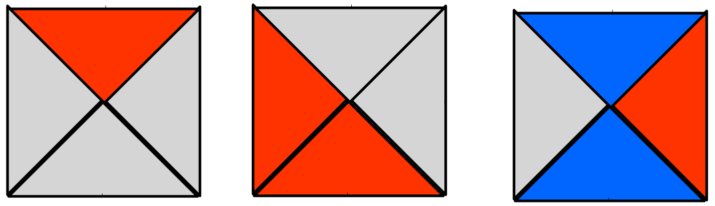


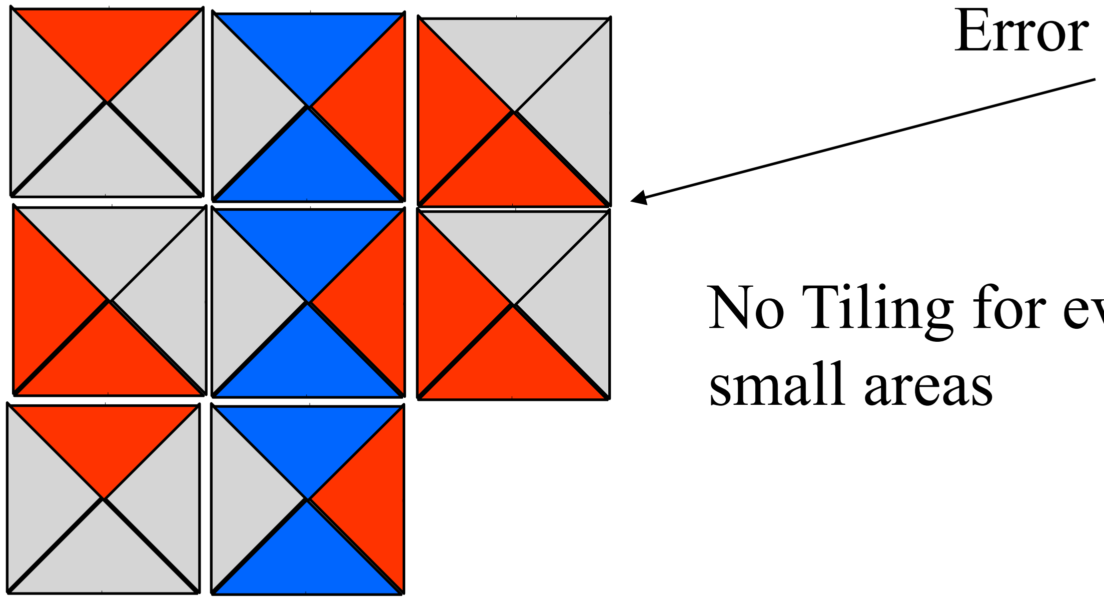


Tiling problem: cover a finite floor

- **Can I use this set of tiles to tile the floor of a house?**
  - **Solvable:** we can try out all combinations
  - Known to be NP-complete problem, complexitywise
- **You can with some sets of tiles, but not all of them**
  - Easy case: white tiles only
  - Not so: next case

Another example tile set:





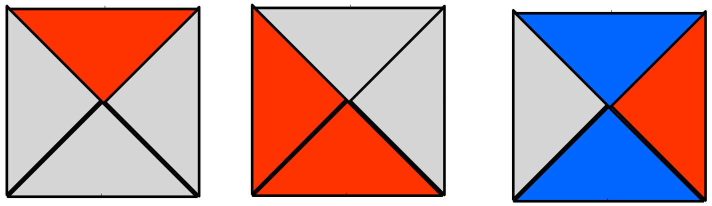
No Tiling for even small areas

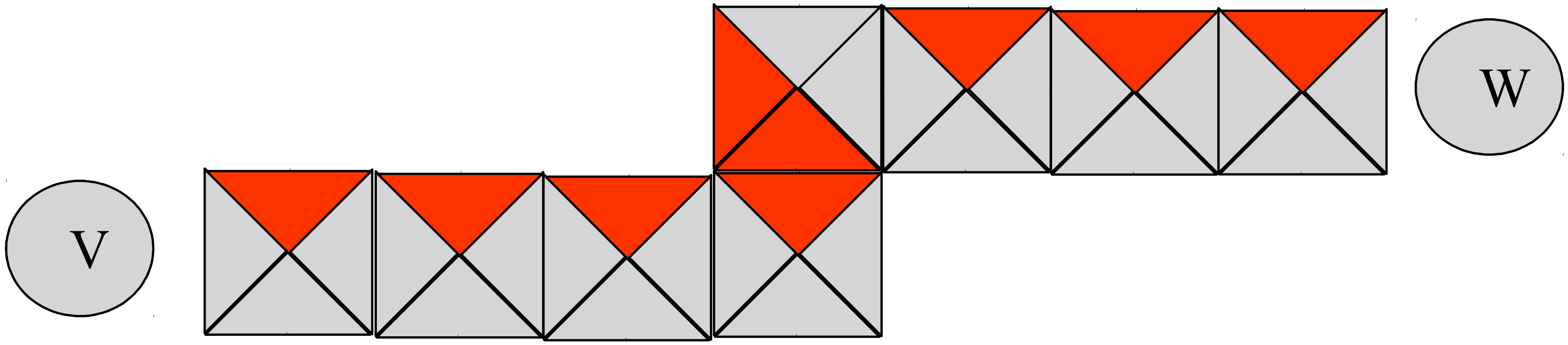
Tiling problem: undecidable question

- **Can a room of any size be tiled using this set of tiles?**
- **Like 2*2, 3*3, 4*4, etc etc ....**
  - **Not solvable!**
  - Cannot try out all cases: there are infinitely many
  - Systematic testing will always find a negative answer
  - No guarantee of finding a positive answer: when to stop testing?

Similar example: “Domino Snakes”

- Is it possible to connect V to W using a tile snake?





For “Domino Snakes” we can prove that:

Let us look at three cases:

- If the plane to lay down the snake in is finite?
  - **trivially decidable**
- If snakes can go anywhere in the plane?
  - **decidable**
- If snakes can only go in half of the plane?
  - **undecidable**

Hilbert’s 10-s problem

- In 1900, mathematician David Hilbert identified 23 mathematical problems and posed them as a challenge for the coming century.

https://en.wikipedia.org/wiki/Hilbert%27s_problems

- **The tenth problem asks for an algorithm to test whether a polynomial has an integral root**
- Apparently, Hilbert assumed that such an algorithm must exist.

https://en.wikipedia.org/wiki/Hilbert%27s_tenth_problem

Problem

- Provide a general algorithm which, for any given Diophantine equation (a polynomial equation with integer coefficients and a finite number of unknowns) can decide whether the equation has a solution with all unknowns taking integer values.
- Examples

3x**2 – 2xy – z*y**2 – 7 = 0

integer solutions: x=1, y=2, z=1

x**2 + y**2 + 1 = 0

no integer solutions

Not solvable

1970 proved by **Yuri Matiyasevich**

Semidecability

- Does X belong to some infinite subset of integers H?
  - Solvable for some subsets H: like “even integers”, “prime integers” etc
  - Not solvable for some H: like integers, for which the corresponding programs halt.

- **Semidecability** means that if X does elong to H, we can find it out by an algorithm. If X does not belong to H, we cannot always show that it does not.

- Halting problem is also semidecidable

- Some undecidable problems are not even semidecidable.

Decidability and Gödel

Hypothese 3 turned out to be false. Gödel showed that 1 and 2 cannot be true at the same time:

[1] Mathematics is consistent. Roughly this means that we cannot prove a statement and its opposite; we cannot prove something horrible like 1=2.

[2] Mathematics is complete. Roughly this means that every true mathematical assertion can be proven i.e. every mathematical assertion can either be proven or disproven.

[3] Mathematics is decidable. This means that for every type of mathematical problem there is an algorithm that, in theory at least, can be mechanically followed to give a solution. We say “in theory” because following the algorithm might take a million years and still be finite

Kurt Gödel

https://de.wikipedia.org/wiki/Datei:1925_kurt_gödel.png

Decidability: Gödel

- In 1930, Kurt Gödel shocked the world by proving that [1] and [2] cannot both be true:

  Either you can prove false statements or there are true statements that are not provable.

- Most people believe that mathematics is incomplete rather than mathematics is inconsistent
- Turing was one of the people who showed that [3] is false by his work on Turing machines and the halting problem
- Important: mathematics is open in the sense that there cannot be a finite set of axioms and rules from which all mathematical truths can be proved.

Decidability: Gödel

- Mathematics is open in the sense that

  No such finite set of axioms and rules can exist from which all true mathematical statatements can be derived

  This holds even for just mathematics dealing with integers

- However, we can always add new axioms and rules. Every true statement can be proven if you add enough axioms and rules.
- Axioms and rules cannot be proven. We can simply believe they are true.

Decidability: Gödel

A simpler form of the incompleteness theorem:

„A complete, consistent and sound axiomatization of all statements about natural numbers is unachievable„

This can be proven relatively easily, using the undecidability of the halting problem:

Assume that we have a consistent and complete axiomatization of all true first-order logic statements about natural numbers. Then we can build an algorithm that enumerates all these statements i.e. an algorithm N that, given a natural number n, computes a true first-order logic statement about natural numbers, such that for all the true statements there is at least one n such that N(n) is equal to that statement.

...

Decidability: Gödel

...

Now suppose we want to decide whether the algorithm with representation a halts on input i. By using Kleene's T predicate, we can express the statement "a halts on input i" as a statement H(a, i) in the language of arithmetic. Since the axiomatization is complete it follows that either there is an n such that N(n) = H(a, i) or there is an n' such that N(n') = Â¬ H(a, i). So if we iterate over all n until we either find H(a, i) or its negation, we will always halt.

This means that this gives us an algorithm to decide the halting problem. Since we know that there cannot be such an algorithm, it follows that the assumption that there is a consistent and complete axiomatization of all true first-order logic statements about natural numbers must be false.