# UNSOLVABILITY AND THE HALTING PROBLEM

## Today

- The halting problem
- Computability and solvability
- Feasibility



## The halting problem

- A *loop* is a set of instructions that repeats several times
- There are 3 kinds of loop
  - counter-controlled
  - condition-controlled
  - forever
- These concepts are the same in any computer programming language!



- Here is an example in computer *pseudo-code*:

```
x=0;
do 3 times
{
  add 1 to x
}
```

- How many times does this loop execute?
- What is the value of `x` when this code completes?

- Another example:

```
x=3;
while ( x > 0 )
{
  subtract 1 from x;
}
```

- How many times does this loop execute?
- What is the value of `x` when this completes?



- And another example:

```
x=1;
while ( x < 5 )
{
  y = x;
}
```

- How many times does this loop execute?
- What is the value of `x` when this completes?



- A program containing an *infinite loop* will run *forever*.
- It will never HALT or TERMINATE.
- This is like setting up a "forever" button in Netlogo.
- When we use a "forever" button, don't know if the program will halt.
  - Netlogo will try to keep the program running once we press the button; but
  - The programmer may have included a "halt" instruction.
- This is called the *halting problem* in computer science.
  - Being able to look at a computer program and determine if it will ever halt (stop).



- Sometimes, whether a program stops or not depends on *input* that it receives.
  - Like your program receiving input from the user.
- here is an example:

```
myProgram( input: x ) {
  while ( x > 0 ) {
    add 1 to x;
  }
}
```

- How many times does this loop execute if `x` = 0?
- How many times does this loop execute if `x` = 1?

## Computability

- A problem is *computable* if it is possible to write a computer program that can solve it.
- A *non-computable* problem is also called *non-solvable*.
- Is the *halting problem* computable?
- In other words, can we write a computer program that will determine if any computer program and its input will halt?
- How would you answer this question?
- Could you try running the program?
  - What if it never halted?



- Suppose we wrote a program ("A") that would take two inputs:
  - Another program "P"; and
  - the input "X" for the other program
- "A" works like this:

  If "P,X" halts, then "A" should run forever.

  If "P,X" does not halt, then "A" should halt.
- The *paradox* is: what if we call program "A" on itself?
- The program cannot produce an answer
  - The program neither halts, nor does it not halt.
- Why?



- This is an example of *proof by contradiction*.
- We assume that a program does exist that can solve the halting problem; then we show that it cannot possibly exist.
- Computability in general is an important question.
- It was considered by concerned mathematicians even before digital computers were developed!
- In the 1930's, much work was devoted to this.
- Ihe Church-Turing thesis (1940's) states basically that any computation that can be defined in an algorithm can be processed on a computer.
- Named after Alonzo Church and Alan Turing.



## Alan Turing

## Feasibility

- Even if a problem is *computable*, it is not always *feasible* to write a program to compute it.
- Sometimes it takes too long to solve a problem.
- A nice example comes from robot soccer:

- A robot might be able to find the ball using a complicated algorithm, but if it takes too long, the soccer game will be over!



- Could you write a program that could count the number of atoms in the universe?
  - The number of atoms is estimated to be about $10^{80}$
- Suppose it took 1 second to count one atom.
- How many seconds would the program need to run to count all of them?
- Here's another way to look at it.
- How many atoms could the program count in a year?
  *num_atoms_per_year*
  $= 60 sec/min \times 60 min/hour \times 24 hours/day \times 365 days/year$
  $= 31,536,000 sec$
  $= 3x10^7$
  how does that compare to $10^{80}$??