CS100: Theory of Computation

Fall 2010

Instructor: James MacGlashan

What is a function?

- A function is a mapping from inputs to outputs
- Takes a set of inputs or parameters
- Produces a single or set of outputs
- Output is generally dependent on input

Simple functions

- Boolean AND
  - AND(a, b) -> c
- Addition
  - Add(a, b) -> c
- Yards to Meters
  - YtoM(y) -> c
- RGB to HSV
  - HSVtoRGB(h,s,v) -> (r, g, b)
- Sorted list
  - Sort({L}) -> {SL}

Determining the mapping

- Computing the function is the process of determining the output from a given input
- Simplest approach is a look up table

Boolean AND

A	B	C
0	0	0
0	1	0
1	0	0
1	1	1

Lookup Tables are insufficient

- How do we design a look up table for yards to meters?
- Better to use a simple algebraic equation

$$m = 0.9144 * y$$

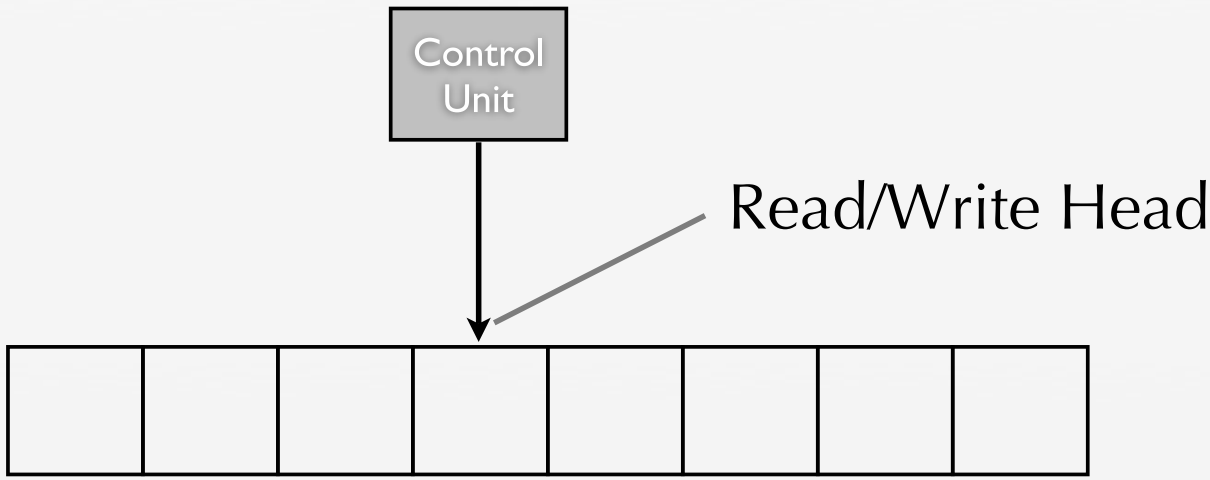
Can we compute any function?

- No! Some function are too complex to compute
- What does this mean for computer scientists?
  - Machines can only perform tasks described by algorithms
  - If a function is not computable, a computer can't "solve" it
- How do we know what is computable?

Turing Machine

- Theoretical computing machine developed by Alan Turing
- Composed of:
  - a tape of cells (can be infinitely long)
    - cells have a finite set of symbols
  - a read/write head
    - for a single step the read/write head can move **one** cell left or right
  - a control unit
    - for which there are a finite set of states; special states for START and HALT
    - The current state coupled with the current symbol dictates next state and head movement

Turing Machine



Turing Binary Add

State = START

*	1	0	1	*

Turing Binary Add

State = START

*	1	0	1	*

Current State	Current Cell	Write Value	Move Direction	Next State
START	*	*	LEFT	ADD
ADD	0	1	RIGHT	RETURN
ADD	1	0	LEFT	CARRY
ADD	*	*	RIGHT	HALT
CARRY	0	1	RIGHT	RETURN
CARRY	1	0	LEFT	CARRY
CARRY	*	1	LEFT	OVERFLOW
OVERFLOW	(any)	*	RIGHT	RETURN
RETURN	0	0	RIGHT	RETURN
RETURN	1	1	RIGHT	RETURN
RETURN	*	*	NO MOVE	HALT

Church-Turing

- A function that is computable by a turing machine is Turing Computable
- Church-Turing thesis states that any computable function is Turing Computable
- Turing machine would be a universal computation machine
- Any other machine that can compute every function a Turing Machine can must be a universal machine as well

Universal Language

- A programming language that can be used to define any Turing-computable function procedure
- Almost all modern languages have high-level complexity/abstraction for convenience, not universality
- Define a very simple language that is a universal language

Bare Bones Language

- Works with only non-negative integers
  - all other data types will be up to the programmer to define in terms of non-negative integers
- Allow for variable names
- End a statement with a ;
- Assignment operators:
  - clear *name*;
    - sets value of name to 0
  - incr *name*;
    - increases value of name by 1
  - decr *name*;
    - decreases value of name by 1 (unless 0)

BBL Control

- One Control Structure
  - while *name* not 0 do;
    .
    .
    .
    end;
- This will execute so long as the variable name does not hold the value 0

Basic procedures

- How do we do direct assignment between variables?

Basic procedures

- How do we do direct assignment between variables?
- Assignment without destruction? (x <- y)

Basic procedures

- How do we do direct assignment between variables?
- Assignment without destruction? (x <- y)
- Adding two numbers? (z <- x + y)

Basic procedures

- How do we do direct assignment between variables?
- Assignment without destruction? (x <- y)
- Adding two numbers? (z <- x + y)
- Multiplying two numbers? (z <- x * y)

Basic procedures

- How do we do direct assignment between variables
- Assignment without destruction? (x <- y)
- Adding two numbers? (z <- x + y)
- Multiplying two numbers? (z <- x * y)
- If-else?
    - e.g., if X not 0 then S1 else S2

Noncomputable functions

- Halting problem is an example of a noncomputable function
- Write a function that can predict whether a function will terminate or not
- Consider BB program:

```
while x not 0 do;
    incr x;
end;
```

- Termination depends on input value of x

Proving the Halting Problem

- We can logically prove that the halting problem is non-computable
- Termination or not depends on input, so we make an special termination case useful for our proof
- A self-terminating program is a program that will terminate if it's input values are set to an encoding of the the program code

Program encoding

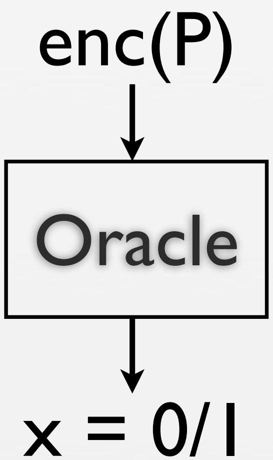
- Look at raw text of a program code
- Take the ASCII value of each character in the program code text and string them together into a single binary value
  - This is a really huge number because program code is usually many many bytes long in text
    - We don't care because this is all theoretical!
- while -> 'w' - 'h' - ...
  'w' = 01110111
  'h' = 01101000
- 0111011101101000...

Proof direction

- Key idea: if we cannot predict whether a program is self-terminating then we cannot compute the halting problem in general
  - since there is at least one input case where we can't: the self-terminating input case

Proof: Step 1

- Propose existence of program, Oracle, that states whether the encoding of another program X is self-terminating
  - that is, if program P sets its input to the value of its program's encoding, program oracle returns a value 1 if it halts, 0 otherwise



x = 0: P is self-terminating
x = 1: P is NOT self-terminating

Proof: Step 2

- Propose a new program, Paradox defined as follows:

enc(P)

↓

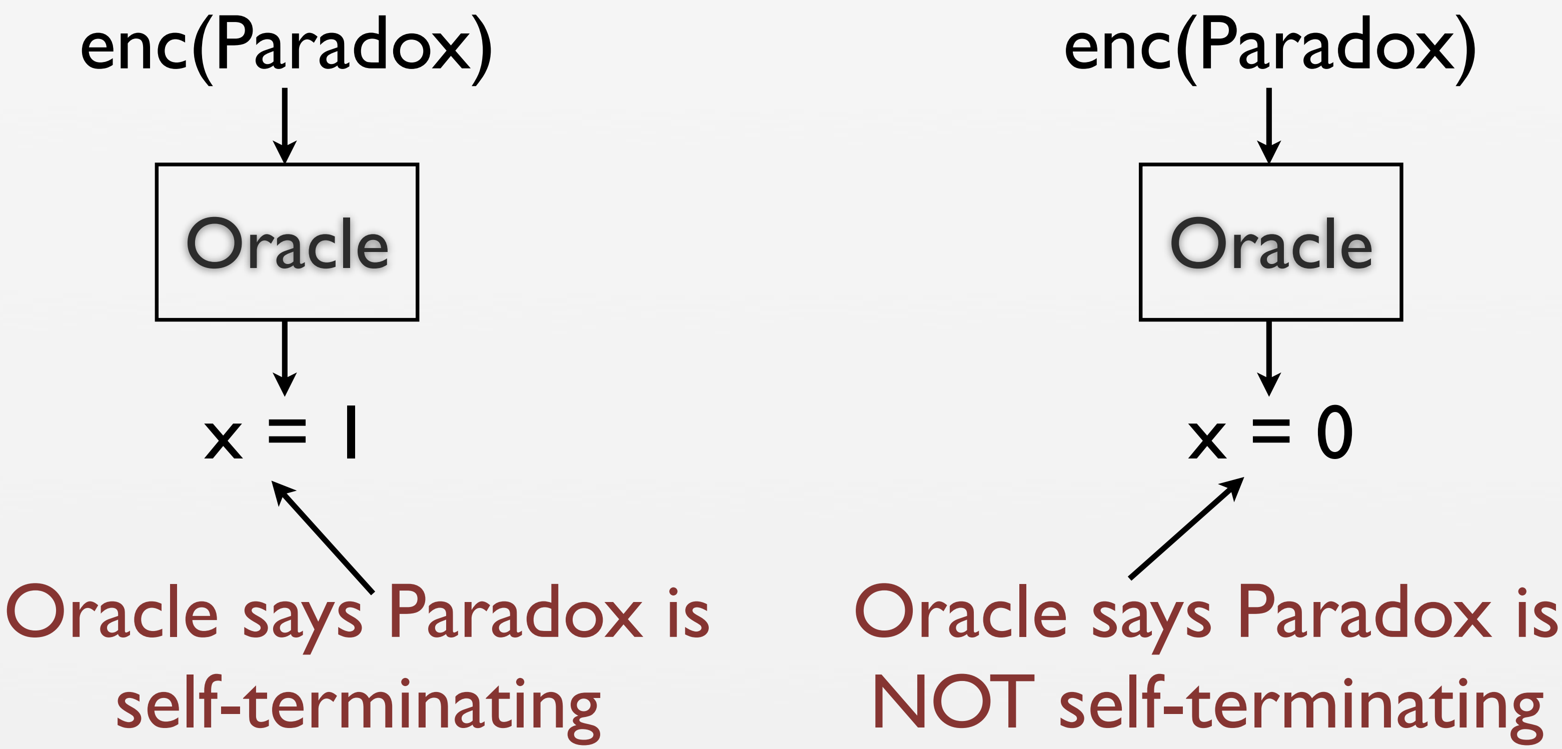
Oracle

↓

```
x = 0/1
while x not 0 do;
    ;
end;
```

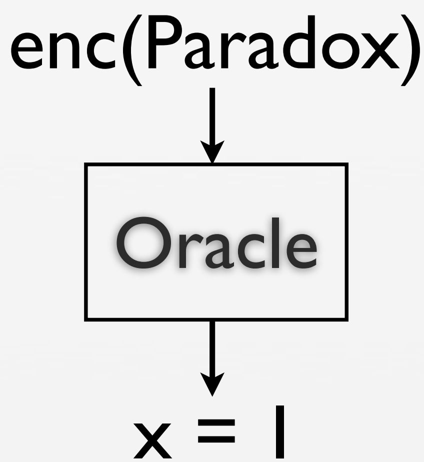
Proof Step 3

- Run Paradox through Oracle; 2 Possibilities

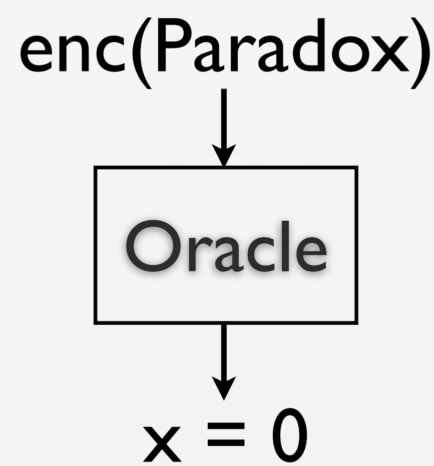


Proof: Step 4

- What happens when we actually run Paradox with it's self encoding as input (self-terminating input)?



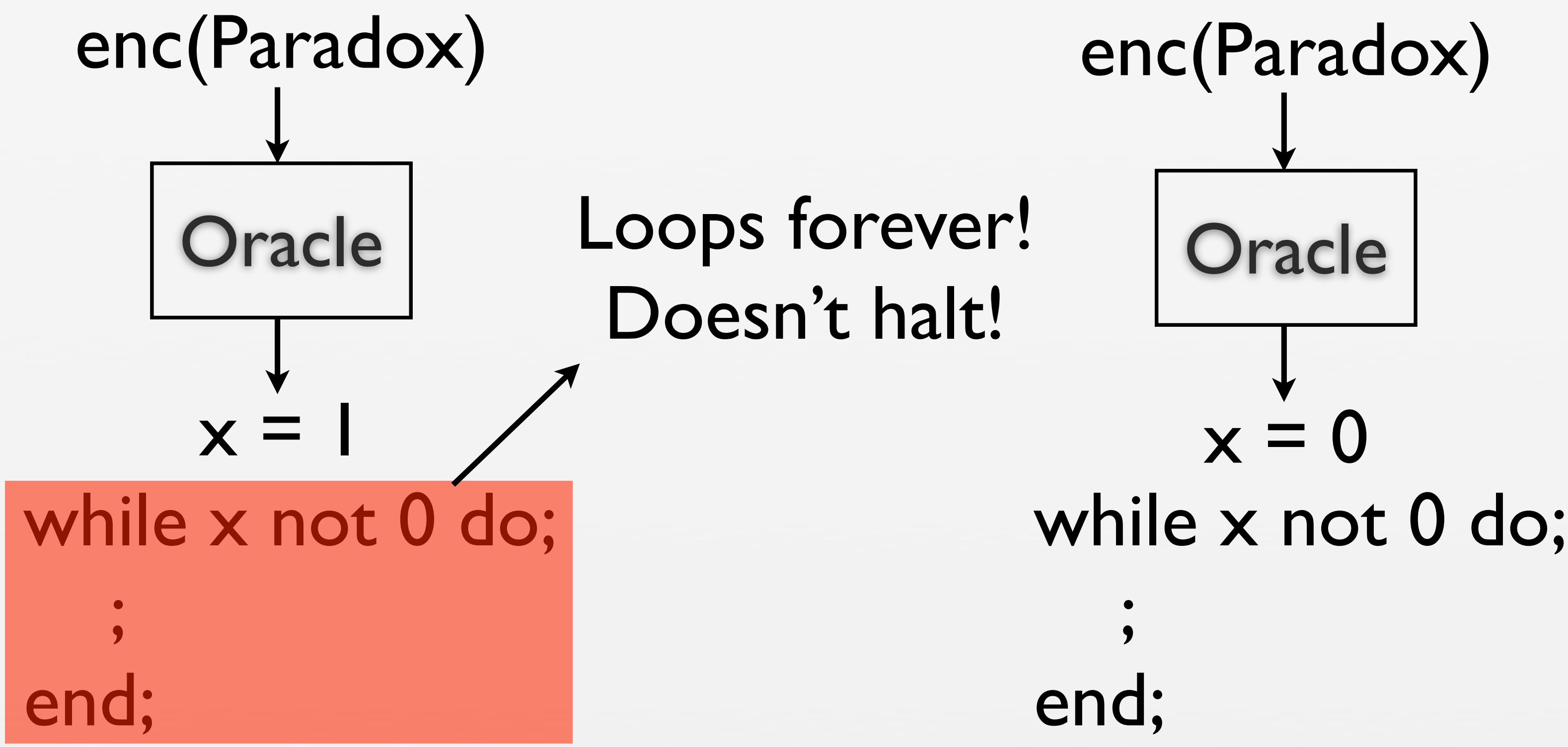

```
while x not 0 do;
    ;
end;
```




```
while x not 0 do;
    ;
end;
```

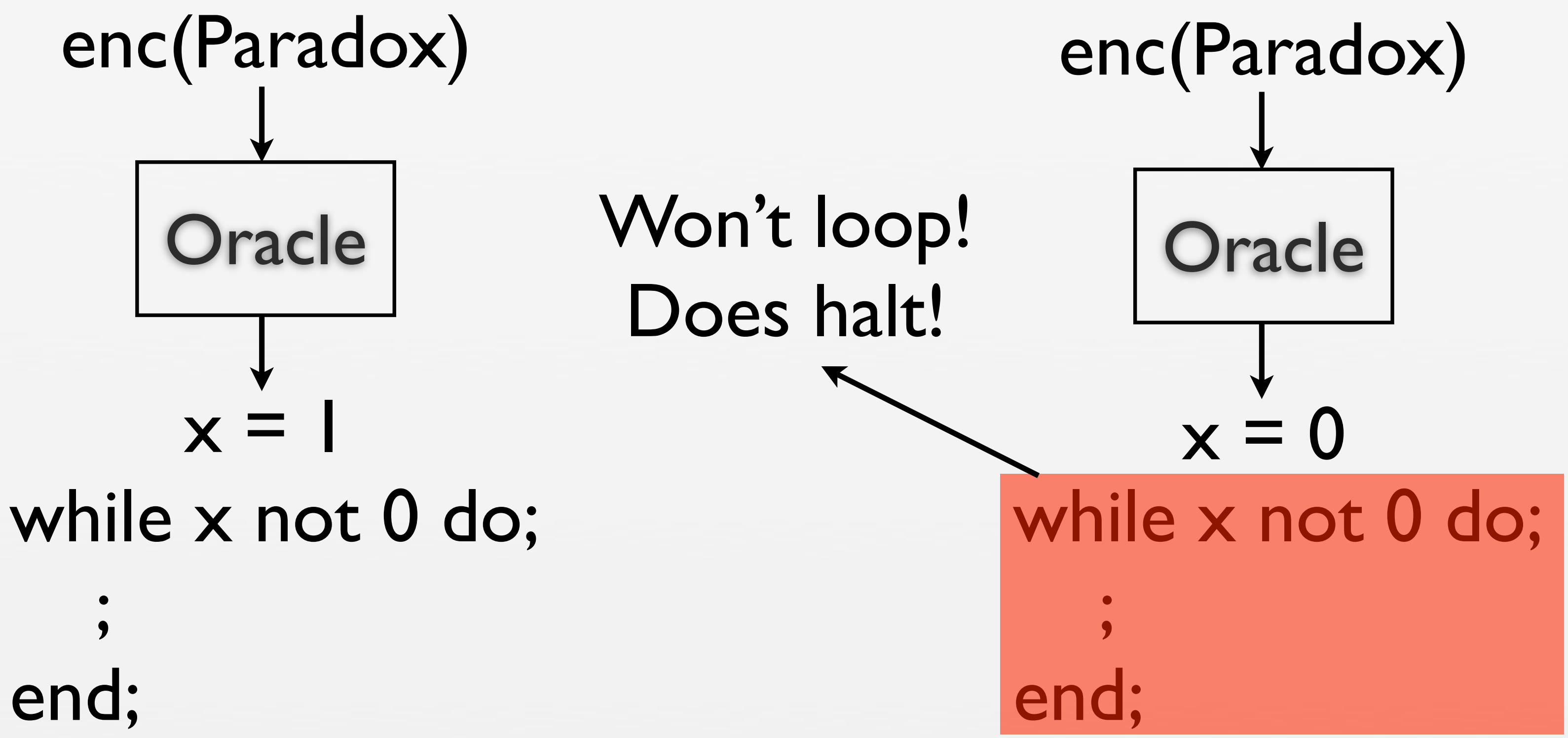
Proof: Step 4

- What happens when we actually run Paradox?



Proof: Step 4

- What happens when we actually run Paradox?



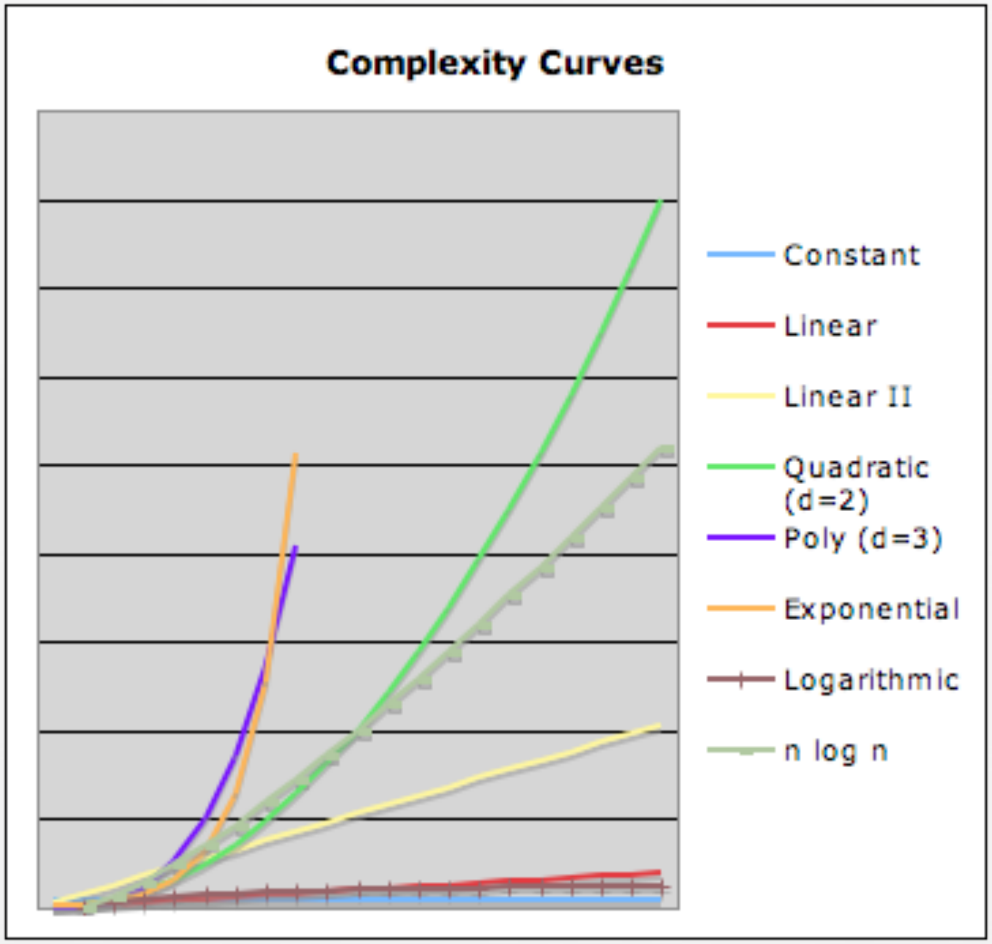
NP-Completeness

- Recall our time complexity analysis?
- Define a measure of how many steps it takes for an algorithm to complete based on the size of its input
- O(f(n)) means that a program's time step complexity is dominated by the function f(n)
  - where n is the input size

Complexity Classes

- O(c): constant; program has a fixed number of steps regardless of input size
- O(log(n)): number steps is dominated by a logarithmic growth in terms of the size of input
- O(n): dominated by linear in terms of n
- O(nlog(n)): dominated by nlogn term
- O(n^c): dominated by polynomial in terms of n
- O(c^n): dominated by exponential in terms of n

Complexity Classes

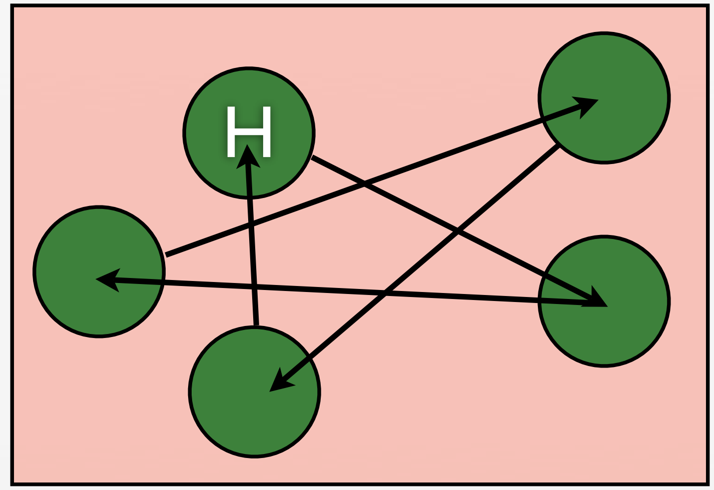


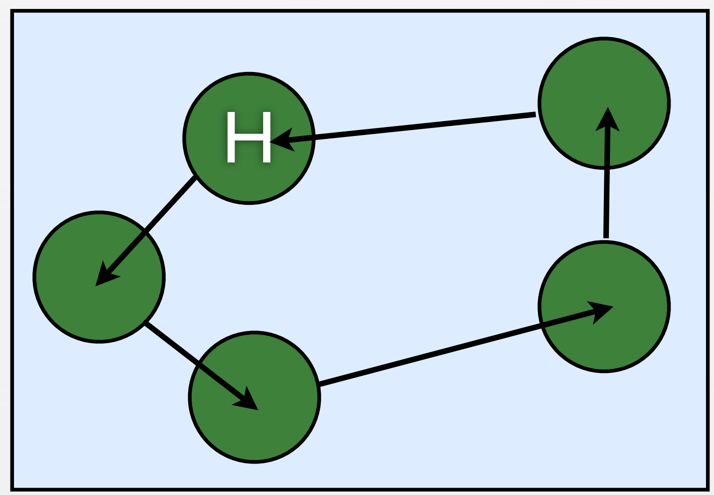
Tractability

- Generally we consider algorithms that can executed in Polynomial time or better to be tractable
  - These grow slowly enough that we can use them on large problems and computer speed will increase until its feasible
  - Exists in complexity class P
- We claim that algorithms that are not bounded by a polynomial term as intractable
  - these grow way to fast to be practical on any large problem

Traveling Salesman (TSP)

- Given a set of n cities each at a different point in space (geography)
- Salesman starts from a home city and must visit each city exactly once and return home
- Salesman has a budget and must find a path that is cheap enough
  - short in number of miles





Determinism and Non-determinism

- An algorithm whose steps are fully defined is deterministic
  - will always execute the same way regardless of executor
  - This is the kind of algorithm in which we program computers
    - random numbers are even deterministic in terms of analysis, but in practice appear random to us
- An algorithm whose steps are not fully defined is non-deterministic
  - Different executors may execute certain steps differently

Algorithms for TSP

- Writing a deterministic algorithm for TSP requires an exponential number of steps
- A non-deterministic algorithm might only take a polynomial number of steps

```
Pick one of the possible paths, p
Compute distance of p, d
if d < allowable milage
    then (declare success)
    else  (declare failure)
```

Algorithms for Traveling Salesman

- Writing a deterministic algorithm for TSP requires an exponential number of steps
- A non-deterministic algorithm might only take a polynomial number of steps

```
Pick one of the possible paths, p
Compute distance of p, d
if d < allowable milage
    then (declare success)
    else  (declare failure)
```

How is a good one picked?

NP Class

- A problem is in the class NP if it can be solved with a non-deterministic polynomial time algorithm
- Any polynomial deterministic algorithm is in NP, but not all NP problems may be in P
  - we strongly believe they are not

NP-Hard

- Class of problems for which we have no known solution in P
- We can prove that a new problem is NP-Hard if we can show that being able to solve it in P would allow us to solve another NP-Hard problems (that are also NP-Complete) in P
  - Transform problems into the settings of others
- An NP-hard problem is at least as hard as the hardest problems in NP

NP-Complete

- Class of problems that both exist in NP and are NP-Hard
- If we ever find a polynomial solution to a single NP-complete problem, then all NP-Complete problems can be solved in Polynomial time
- We strongly believe this cannot be done, but we have no proof
- Prove $P = NP$ or $P \neq NP$
  - You'll be hugely famous, and if you prove the former, you'll have changed everything!