Theory of Computation

Theory of Computation

- What is possible to compute?
- We can prove that there are some problems computers cannot solve
- There are some problems computers can theoretically solve, but are intractable (would take too long to compute to be practical)

Automata Theory

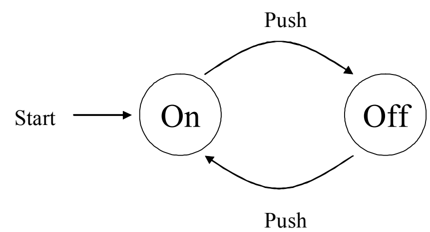
- The study of abstract computing devices, or "machines."
- Days before digital computers
  - What is possible to compute with an abstract machine
  - Seminal work by Alan Turing
- Why is this useful?
  - Direct application to creating compilers, programming languages, designing applications.
  - Formal framework to analyze new types of computing devices, e.g. biocomputers or quantum computers.
- Covers simple to powerful computing "devices"
  - Finite state automaton
  - Grammars
  - Turing Machine

Finite State Automata

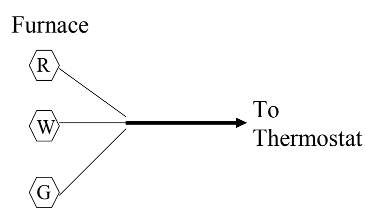
- Automata – plural of "automaton"
  - i.e. a robot
- Finite state automata then a "robot composed of a finite number of states"
  - Informally, a finite list of states with transitions between the states
- Useful to model hardware, software, algorithms, processes
  - Software to design and verify circuit behavior
  - Lexical analyzer of a typical compiler
  - Parser for natural language processing
  - An efficient scanner for patterns in large bodies of text (e.g. text search on the web)
  - Verification of protocols (e.g. communications, security).

On-Off Switch Automaton

- Here is perhaps one of the simplest finite automaton, an on-off switch
- States are represented by circles. Edges or arcs between states indicate transitions or inputs to the system. The “start” edge indicates which state we start in.
- Sometimes it is necessary to indicate a “final” or “accepting” state. We’ll do this by drawing the state in double circles



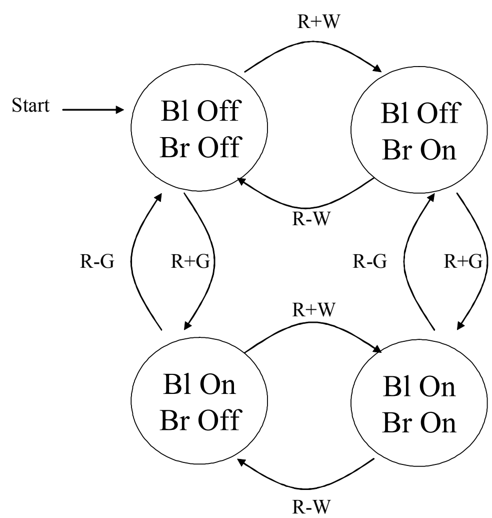
Gas Furnace Example



- The R terminal is the hot wire and completes a circuit. When R and G are connected, the blower turns on. When R and W are connected, the burner comes on. Any other state where R is not connected to either G or W results in no action.

Furnace Automaton

- Could be implemented in a thermostat

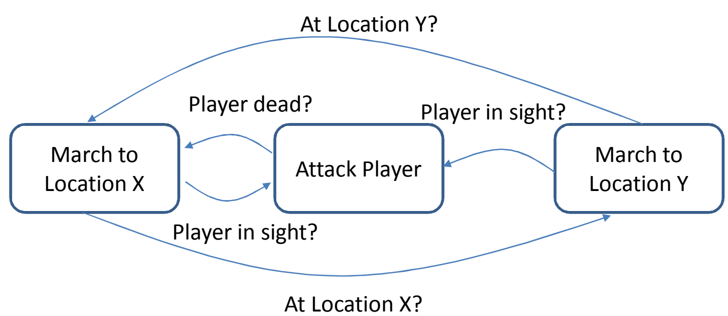


Furnace Notes

- We left out connections that have no effect
  - E.g. connecting W and G
- Once the logic in the automata has been formalized, the model can be used to construct an actual circuit to control the furnace (i.e., a thermostat).
- The model can also help to identify states that may be dangerous or problematic.
  - E.g. state with Burner On and Blower Off could overhead the furnace
  - Want to avoid this state or add some additional states to prevent failure from occurring (e.g., a timeout or failsafe)

Often Used for Video Game AI

- Computer AI player for a sentry guarding two locations



Example

- Design a finite state automaton that determines if some input sequence of bits has an odd number of 1's

Grammars

- Grammars provide a different "view" of computing than automata
  - Describes the "Language" of what is possible to generate
  - Often grammars are identical to automata
- Example: Odd finite state automaton
  - Could try to describe a grammar that generates all possible sequences of 1's and 0's with an odd number of 1's

Grammar Example

- Just like English, languages can be described by grammars. For example, below is a very simple grammar:

  S → Noun Verb-Phrase
  Verb-Phrase → Verb Noun
  Noun → { Kenrick, cows }
  Verb → { loves, eats }

- Using this simple grammar our language allows the following sentences. They are "in" the Language defined by the grammar:

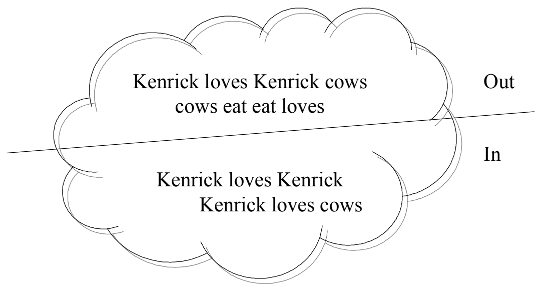
  Kenrick loves Kenrick
  Kenrick loves cows
  Kenrick eats Kenrick
  Kenrick eats cows
  Cows loves Kenrick
  Cows loves cows
  Cows eats Kenrick
  Cows eats cows

- Some sentences not in the grammar:

  Kenrick loves cows and kenrick.
  Cows eats love cows.
  Kenrick loves chocolate.

Grammars and Languages

- The "sentences" that a grammar generates can describe a particular problem or solution to a problem
- Grammar provides a "cut" through the space of possible sentences – can be crude to sophisticated cuts
- Grammars can represent languages that deterministic finite automaton cannot



Grammar Example

- What can the following grammar generate?

S → 0
S → 0S1

- What can the following grammar generate?

S → 1
S → Z1ZSZ1Z
Z → empty
Z → 0
Z → 0Z

Taxonomy of Complexity

Uncomputable	
Turing Machines	Phrase Structure
Linear bounded automata	Context-Sensitive
Pushdown automata	Context-Free
Finite state automata	Regular
Machines	Grammars/Languages

Complex

↑

Crude

Turing Machine

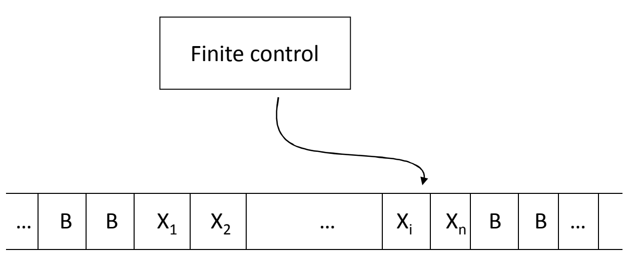
- Finite state automatons and grammars have limitations for even simple tasks, too restrictive as general purpose computers
- Enter the **Turing Machine**
  – More powerful than either of the above
  – Essentially a finite state automaton but with unlimited memory
  – Although theoretical, can do everything a general purpose computer of today can do
    - If a TM can't solve it, neither can a computer

Turing Machines

- TM's described in 1936
  - Well before the days of modern computers but remains a popular model for what is possible to compute on today's systems
  - Advances in computing still fall under the TM model, so even if they may run faster, they are still subject to the same limitations
- A TM consists of a finite control (i.e. a finite state automaton) that is connected to an infinite tape.

Turing Machine

- The tape consists of cells where each cell holds a symbol from the tape alphabet. Initially the input consists of a finite-length string of symbols and is placed on the tape. To the left of the input and to the right of the input, extending to infinity, are placed blanks. The tape head is initially positioned at the leftmost cell holding the input.



Turing Machine Details

- In one move the TM will:
  - Change state, which may be the same as the current state
  - Write a tape symbol in the current cell, which may be the same as the current symbol
  - Move the tape head left or right one cell
  - The special states for rejecting and accepting take effect immediately

A Turing machine for incrementing a value

Current state	Current cell content	Value to write	Direction to move	New state to enter
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	(Ignored)	*	Right	RETURN
RETURN	~~1~~ 0	0	Right	RETURN
RETURN	~~*~~ 1	1	Right	RETURN
RETURN	*	*	No move	HALT

Equivalence of TM's and Computers

- In one sense, a real computer has a finite amount of memory, and thus is **weaker** than a TM.
- But, we can postulate an infinite supply of tapes, disks, or some peripheral storage device to simulate an infinite TM tape. Additionally, we can assume there is a human operator to mount disks, keep them stacked neatly on the sides of the computer, etc.
- Need to show both directions, a TM can simulate a computer and that a computer can simulate a TM

Computer Simulate a TM

- This direction is fairly easy - Given a computer with a modern programming language, certainly, we can write a computer program that emulates the finite control of the TM.
- The only issue remains the infinite tape. Our program must map cells in the tape to storage locations in a disk. When the disk becomes full, we must be able to map to a different disk in the stack of disks mounted by the human operator.

TM Simulate a Computer

- In this exercise the simulation is performed at the level of stored instructions and accessing words of main memory.
  - TM has one tape that holds all the used memory locations and their contents.
  - Other TM tapes hold the program counter, memory address, computer input file, and scratch data.
  - The computer's instruction cycle is simulated by:

1. Find the word indicated by the program counter on the memory tape.
2. Examine the instruction code (a finite set of options), and get the contents of any memory words mentioned in the instruction, using the scratch tape.
3. Perform the instruction, changing any words' values as needed, and adding new address-value pairs to the memory tape, if needed.

TM/Computer Equivalence

- Anything a computer can do, a TM can do, and vice versa
- TM is much slower than the computer, though
  - But the difference in speed is polynomial
  - Each step done on the computer can be completed in $O(n^2)$ steps on the TM
- While slow, this is key information if we wish to make an analogy to modern computers. Anything that we can prove using Turing machines translates to modern computers with a polynomial time transformation.
- Whenever we talk about defining algorithms to solve problems, we can equivalently talk about how to construct a TM to solve the problem. If a TM cannot be built to solve a particular problem, then it means our modern computer cannot solve the problem either.

Church-Turing Thesis

- The functions that are computable by a Turing machine are exactly the functions that can be computed by any algorithmic means.

Universal Programming Language

A language with which a solution to any computable function can be expressed

- Examples: “Bare Bones” and most popular programming languages

The Bare Bones Language

- Bare Bones is a simple, yet universal language.
- Statements
  - **clear *name*;**
  - **incr *name*;**
  - **decr *name*;**
  - **while *name* not 0 do; … end;**

A Bare Bones program for computing $X \times Y$

```
clear Z;
while X not 0 do;
   clear W;
   while Y not 0 do;
      incr Z;
      incr W;
      decr Y;
   end;
   while W not 0 do;
      incr Y;
      decr W;
   end;
   decr X;
end;
```

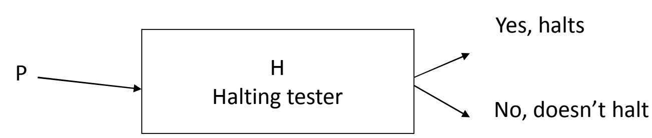
A Bare Bones implementation of the instruction “copy Today to Tomorrow”

```
clear Aux;
clear Tomorrow;
while Today not 0 do;
   incr Aux;
   decr Today;
end;
while Aux not 0 do;
   incr Today;
   incr Tomorrow;
   decr Aux;
end;
```

The Halting Problem

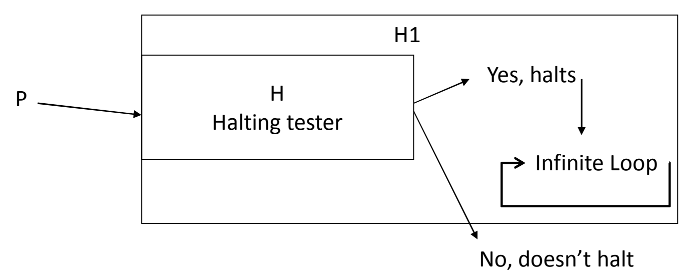
- Given the encoded version of any program, return 1 if the program is self-terminating, or 0 if the program is not.
- First thought: Run the program to see if it halts or not. Problem?

Halting Tester



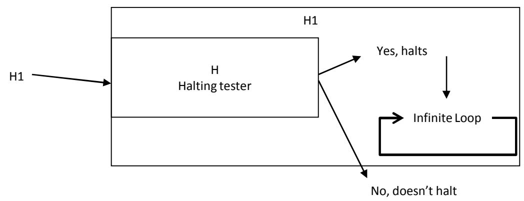
Halting Tester (2)

Next we modify H to a new program H1 that acts like H, but when H prints "Yes, halts", H1 enters an infinite loop



Halting Tester (3)

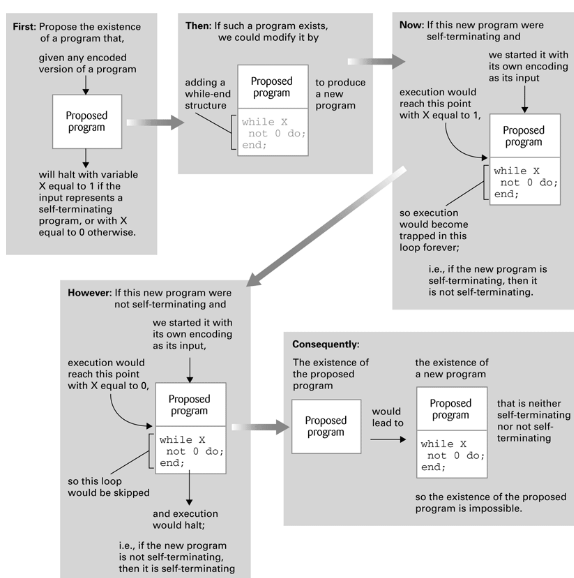
- However, H1 cannot exist. If it did, what would H1(H1) do?
- That is, we give H1 as input to itself:



If H1 on the left halts, then H1 given H1 as input will enter an infinite loop and not halt, in which case it should output that it doesn't halt. But we just supposed that H1 is supposed to halt.

The situation is paradoxical and we conclude that H1 cannot exist and this problem is **undecidable**.

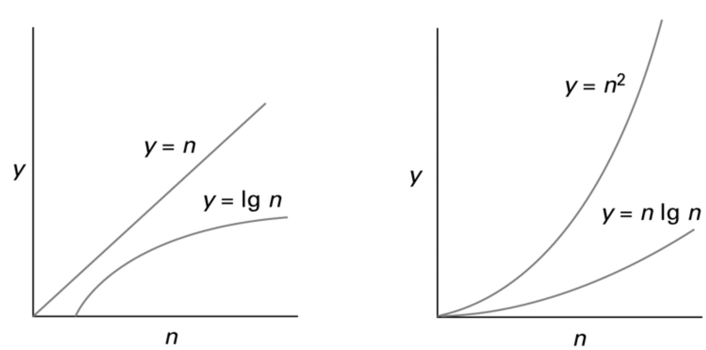
The Halting program is unsolvable



Complexity of Problems

- **Time Complexity:** The number of instruction executions required
  - Unless otherwise noted, "complexity" means "time complexity."
- Theta or Big-O notation
  - A problem is in class $\Theta(f(n))$ if it can be solved in some number of steps proportional to f(n)
  - A problem is in class $O(f(n))$ if it can be solved in some number of steps proportional or less than f(n); i.e. f(n) is an upper bound
- Examples
  - Sequential search is $\Theta(n)$
  - Binary search is $\Theta(\lg n)$
  - Insertion Sort is $O(n^2)$

Graphs of the mathematical expressions *n*, lg *n, n* lg n, and n^2



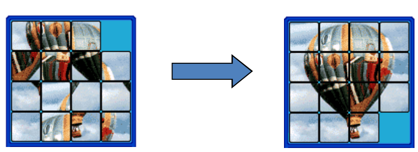
a. *n* versus lg *n* **b.** n^2 versus *n* lg *n*

P versus NP

- **Class P:** All problems in any class $\Theta(f(n))$, where f(n) is a polynomial; problem can be solved in polynomial time
- **Class NP:** All problems that can be solved by a nondeterministic algorithm in polynomial time
  - **Nondeterministic algorithm** = an algorithm described by a Turing Machine that could be in multiple states at the same time
  - Given a proposed solution to a problem, can verify if the proposed solution is an actual solution in polynomial time.
- Whether the class NP is bigger than class P is currently unknown.

NP $\supseteq$ P

- NP is obviously a superset of P
- But many problems appear to be in NP but not in P
  - E.g., consider a "sliding tile" puzzle



Solve in polynomial time? (e.g. function of # of tiles)
But given a proposed solution, easy to verify if it is correct in polynomial time

- 29 Node Traveling Salesperson Problem
- 29! = 8.8 trillion billion billion possible asymmetric routes.
- ASCI White, an IBM supercomputer being used by Lawrence Livermore National Labs to model nuclear explosions, is capable of 12 trillion operations per second (TeraFLOPS) peak throughput
- Assuming symmetric routes, ASCI White would take 11.7 billion years to exhaustively search the solution space

The Big Question

- Is there anything in NP that is not in P?
- We know that P $\subseteq$ NP
- But it is unknown if P = NP
- Most people believe that P $\neq$ NP due to the existence of problems in NP that are in the class NPC, or NP Complete

- The Clay Mathematics Institute has offered a million dollar prize to anyone that can prove that P=NP or that P$\neq$NP