So… what can’t computers do?

- (Or, can we summarize what can they do?)
- Given all that we’ve learned this semester, it’s actually pretty hard to characterize
- Focus of computation theory is to determine what is computable and what is not
  - Computable implies functions whose output values can be determined algorithmically from their input values
  - So, what’s an example of a noncomputable function?
Formalizing computability

- Several popular ways
  - (Finite) state machines
  - Turing machines
- State machines are a sort of like a flowchart
  - One starts at a “start state”, goal is to get to the “end” or “goal” state
  - State transitions specify what to do based on initial input
  - States represent the “current” computer’s state
  - States implicitly store what has happened before.
  - Problem: intermediate storage?

Example of FSM

- locate the string "abba" in a file containing "aababbbabba..."
- Input: a a b a b b b a b b a
- State: 0 1 1 2 1 2 3 0 1 2 3 4
Turing machine

- A state machine on steroids
- Idea: not only do we have state, but we have storage
- Alan Turing modeled the storage as a "paper tape" in 1936
- The tape is manipulated by a read/write head that can move left and right one space

![Turing Machine Diagram]

Turing Machine’s Computation

- A Turing machine’s computation consists of a sequence of steps that are executed by the control unit

- Each step consists of
  - Observing the symbol in the current tape cell
  - Writing a symbol in that cell
  - Possibly moving the read/write head one cell to the left or right
  - Changing states

- The exact action to be performed is determined by a program that tells the control unit what to do based on the machine’s state and the content of the current tape cell
Simple Turing example

- Add one to a number already encoded on tape
- We encode it as a binary number, and surround it with the start/end states ("*"
- Let's do this...

<table>
<thead>
<tr>
<th>Current state</th>
<th>Current cell content</th>
<th>Value to write</th>
<th>Direction to move</th>
<th>New state to enter</th>
</tr>
</thead>
<tbody>
<tr>
<td>START</td>
<td>*</td>
<td>*</td>
<td>Left</td>
<td>ADD</td>
</tr>
<tr>
<td>ADD</td>
<td>0</td>
<td>1</td>
<td>Right</td>
<td>RETURN</td>
</tr>
<tr>
<td>ADD</td>
<td>1</td>
<td>0</td>
<td>Left</td>
<td>CARRY</td>
</tr>
<tr>
<td>ADD</td>
<td>*</td>
<td>*</td>
<td>Right</td>
<td>HALT</td>
</tr>
<tr>
<td>CARRY</td>
<td>0</td>
<td>1</td>
<td>Right</td>
<td>RETURN</td>
</tr>
<tr>
<td>CARRY</td>
<td>1</td>
<td>0</td>
<td>Left</td>
<td>CARRY</td>
</tr>
<tr>
<td>CARRY</td>
<td>*</td>
<td>1</td>
<td>Left</td>
<td>OVERFLOW</td>
</tr>
<tr>
<td>OVERFLOW</td>
<td>*</td>
<td>*</td>
<td>Right</td>
<td>RETURN</td>
</tr>
<tr>
<td>RETURN</td>
<td>0</td>
<td>0</td>
<td>Right</td>
<td>RETURN</td>
</tr>
<tr>
<td>RETURN</td>
<td>1</td>
<td>1</td>
<td>Right</td>
<td>RETURN</td>
</tr>
<tr>
<td>RETURN</td>
<td>*</td>
<td>*</td>
<td>No move</td>
<td>HALT</td>
</tr>
</tbody>
</table>

A Turing Machine Example (1/2)

Machine State = START

Current Position

Machine State = ADD

Current Position

Machine State = CARRY

Current Position
A Turing Machine Example (2/2)

So why bother with Turing?

- Church-Turing thesis: the set of Turing functions is the same as the set of functions that are computable in general!
  - Although some may look really awkward in a Turing machine
- Widely accepted by computer scientists today
- A language is Turing-complete if it can encode all that a Turing machine can do
  - Both C and Java are Turing-complete
Noncomputability, redux

- So, noncomputable functions can’t be modeled as a Turing machine
- How do we demonstrate?
  - Not that trivial, beyond scope of class
- Most famous noncomputable function: *Will a specified program halt?*

Universal Programming Language

- Most features in today’s high-level languages merely enhance *convenience* rather than contribute to the fundamental *power* of the language
- Our approach here is to describe a simple imperative programming language powerful enough to express programs for computing all the computable functions
- A programming language with this power is called a *universal* programming language
- The language we present is quite simple; we call it Bare Bones in that it isolates a *minimal* set of requirements of a universal programming language
The Bare Bones Languages

- All variables are considered to be of type “bit pattern of arbitrary length”
- Variable names must begin with a letter, which can be followed by any combination of letters and digits
- Contains three assignment statements and one loop structure
  - clear name;
  - incr name;
  - decr name;
  - while name not 0 do;
  - .
  - .
  - end;

Programming in Bare Bones

```
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;
```

A Bare Bones program for “copy Today to Tomorrow”

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

A Bare Bones program for computing X * Y
The Universality of Bare Bones

- Researchers have shown that the Bare Bones language can be used to express algorithms for computing all the Turing-computable functions.
- That is, any computable function can be computed by a program written in Bare Bones.
- Thus Bare Bones is a universal programming language; if an algorithm exists for solving a problem, then that problem can be solved by some Bare Bones program.
- Bare Bones could theoretically serve as a general-purpose programming language.

Halting Problem

- The halting problem is the problem of trying to predict in advance whether a program will terminate if started under certain conditions.
- Consider the simple Bare Bones program:

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

- If the initial value of X is 0, then the program will halt; otherwise, the loop will be executed forever.
- It is easy in the above example to predict a program’s behavior; however, this task may be more complicated or even impossible in some cases.
Self-Reference

- Whether a program ultimately halts can depend on the initial values of its variables.
- We assign a program’s variables an initial value representing the program itself; that is, we assign the encoded version of a program as the value of its variables.

Self-Termination

- A Bare Bones program is self-terminating if its execution terminates when started with itself as its input.
- The halting problem is now precisely described as the problem of determining whether Bare Bones programs are or are not self-terminating.
- There is no algorithm for answering this question in general; thus, the solution to the halting problem lies beyond the capabilities of computers.
Unsolvability of the Halting Problem (1/3)

**First:** Propose the existence of a program that, given any encoded version of a program, will halt with variable X equal to 1 if the input represents a self-terminating program, or with X equal to 0 otherwise.

**Then:** If such a program exists, we could modify it by adding a while-end structure to produce a new program.

Unsolvability of the Halting Problem (2/3)

**Now:** If this new program were self-terminating and we started it with its own encoding as its input, execution would reach this point with X equal to 1, so execution would become trapped in this loop forever; i.e., if the new program is self-terminating, then it is not self-terminating.
Unsolvability of the Halting Problem (3/3)

However: If this new program were not self-terminating and we started it with its own encoding as its input, execution would reach this point with \( X \) equal to 0, so this loop would be skipped and execution would halt; i.e., if the new program is not self-terminating, then it is self-terminating.

Consequently:
The existence of the proposed program would lead to the existence of a new program that is neither self-terminating nor not self-terminating.

Therefore, the halting problem is unsolvable.

Complexity of Problems (1/2)

- We are interested in the question of whether a solvable problem has a practical solution.
- The complexity of a problem is determined by the properties of the algorithms that solve that problem.
- More precisely, the complexity of the simplest algorithm for solving a problem is considered to be the complexity of the problem itself.
- We measure an algorithm’s complexity in terms of the time required for its execution, which is proportional to the number of steps that must be performed.
Complexity of Problems (2/2)

- The complexity of a problem is $\Theta(f(n))$ if there is an algorithm with complexity of $\Theta(f(n))$ for solving the problem and no other algorithm has a lower complexity.

- Finding the best solution to a problem and knowing that it is the best is often a difficult problem itself; in such situations, big O notation is used.

- The complexity of a problem is $O(f(n))$ if it has a solution whose complexity is $\Theta(f(n))$ but it could possibly have a better solution.

Polynomial vs. Nonpolynomial Problems

- “$g(n)$ is bounded by $f(n)$” means that the graph of $f(n)$ will be above the graph of $g(n)$ for “large” values of $n$.

- A problem is a polynomial problem if the problem is in $O(f(n))$, where the expression $f(n)$ is either a polynomial itself or bounded by a polynomial.

- The collection of all polynomial problems is denoted by $P$.

- Problems that are outside the class $P$ are characterized as having extremely long execution times.

- Identifying the problems that belong to $P$ is of major importance in computer science because it tells whether problems have practical solutions.
Classes of computable functions

- We typically break them down by the time they take to run; here are some typical values that we’ve seen:
  
  ![Graphs](#)

  a. $n$ versus $\lg n$
  b. $n^2$ versus $n \lg n$

So...

- We call such functions for which we know no better way to be “nondeterministic polynomial”, or NP
  - Typically exponential
- We care because lots of useful problems fall into this category

![Diagram](#)
In fact, NP is “useful”

- Public-key encryption (e.g., SSL/ssh) largely works on the fact that decrypting an encrypted message takes an extraordinarily long time
  - Details beyond scope of class
- If someone were to prove that P=NP, many of today’s encryption algorithms would have to be thrown out the window
- Fortunately, no one has come close to proving it
- But no one has come close to proving the opposite either

So where do we go from here?

- Most computer scientists (except great theoreticians) focus on making new computable algorithms, hopefully in polynomial time
- With the knowledge you’ve learned in this class, you have the pieces to go ahead and build such algorithms, and code them
- Remaining CS classes introduce advanced concepts, but they still boil down to the same thing
Thank you!

- You guys have been a great audience.
- I hope you found this class rewarding.
- Good luck with the rest of your Computer Science mini-careers!
  - And with final