Greatest Hits

April 25–26, 2012
CS 60: Principles of Computer Science

Monday, April 30: Student Presentations

11–11:30am, GA Edwards: Computer Animation Projects

Tuesday, May 1: Clinic Projects Day

10:30am–1pm, Platt: Poster Session

Wednesday, May 2: Student Presentations

4–5:30pm, Platt: Poster Session

Final Exam Info

3 hours.

Closed-book, except for 2 handwritten sheets.

Available for pickup by Monday, May 7.

Due back by 5pm on Friday, May 11.

Cumulative, with an emphasis on the second half.

Review sheet posted on the course web page.

Course Comparison

CS 5 Goals:

- ✓ Solving computational problems in *one* language
- ✓ Conveying some of CS's *breadth*

CS 60 Goals:

- ✓ Solving computational problems in *several* languages
- ✓ Conveying more of CS's *depth*
- ✓ More emphasis on clarity, efficiency, and limits of computation.
- ✓ Strengthening programming skills (prep for CS 70)

What is This?

01101100011010010110011001100101

Sapir-Whorf Hypothesis


(Not this Worf)

"The language we use determines the way in which we view and think about the world" —Sapir and Whorf (1950's)

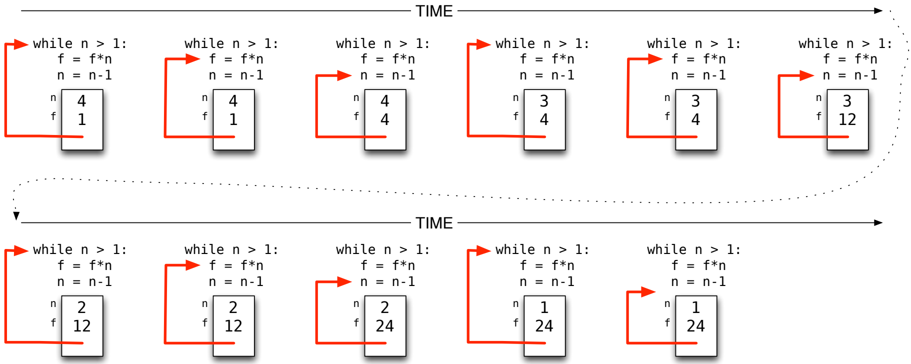
"A language that doesn't affect how you think about programming isn't worth knowing." — Alan Perlis

"For application software, you want to be using the most powerful (reasonably efficient) language you can get...[yet programmers are] satisfied with whatever language they happen to use, because it dictates the way they think about programs." — Paul Graham

Imperative Programming

Step-by-step instructions for updating memory (data)

```
while n > 1:
    f = f*n
    n = n-1
```



Functional Programming

Calculating answers (in terms of sub-calculations)

```
(define (fac n)
   (if (equal? n 1)
       1
       (* n (fac (- n 1)))))
```

```
∴   (fac 4)  =  (* 4 (fac 3))
             =  (* 4 (* 3 (fac 2)))
             =  (* 4 (* 3 (* 2 (fac 1))))
             =  (* 4 (* 3 (* 2 (* 1 (fac 0)))))
             =  (* 4 (* 3 (* 2 (* 1 1))))
             =  (* 4 (* 3 (* 2 1)))
             =  (* 4 (* 3 2))
             =  (* 4 6)
             =  24
```

Logic Programming

Describe a solution, not the process.

```
# Prolog Example
#   Assumes permutation and increasing
#   were previously defined

sort(In,Out) :- permutation(In,Out), increasing(Out).

# Then we can ask...

?- sort([3,6,2,1], Answer).
Answer = [1, 2, 3, 6].
```

Other Useful Operations, by Example

```
(null? '(1 2 3))          ;; ===>  #f
(null? '()      )         ;; ===>  #t

(second '(1 2 3))         ;; ===>  2
(third  '(1 2 3))         ;; ===>  3

(length '(1 2 3))         ;; ===>  3

(append '(1 2) '(3))      ;; ===> '(1 2 3)

(list 1 2 (+ 1 2))        ;; ===> '(1 2 3)

'(1 2 (+ 1 2))            ;; ===> '(1 2 (+ 1 2))
```

In the world of Big-O

- ✓ *Constant* factors are ignored
- ✓ Inputs are arbitrarily large (so "small" sums are ignored)
- ✓ We are looking for upper bounds.

$O(1)$:
- 6 steps
- 1 (big?) step
- no more than 4000 steps
- somewhere between 2 and 47 steps

$O(n)$:
- $100n + 3$ steps
- $n - 1$ (big?) steps
- anywhere between 3 and 69 steps per item, for n items.

$O(n^2)$:
- $2n^2 + 100n + 3$ steps
- $n^2 - n$ (big?) steps
- somewhere between 1 and 40 steps per item, for n^2 items
- anywhere between $\log n$ and $7n$ steps per item, for n items.

Checking Membership in a List

```
; Given a value e and a list L, check if e is in L
 (define (member e L)
   (cond
       [ (null? L)                #f ]
       [ (equal? e (first L))     #t ]
       [ else                     (member e (rest L)) ]))
```

Worst-case asymptotic running time?

Some "Anonymous Functions" in Racket

- ✓ The "successor function" (Does the name x matter?)

```
(lambda (x) (+ x 1))
```

- ✓ The "geometric mean" function

```
(lambda (x y) (sqrt (* x y)))
```

- ✓ The "is-greater-than-5 function"

```
(lambda (N) (> N 5))
```

- ✓ The "is-a-list-of-length-two function"

```
(lambda (L) (and (list? L) (= (len L) 2)))
```

- ✓ The squaring function?

map: A Recursion Alternative

```
;; apply function f to all the elements in L
(define (map f L)
  (if (null? L)
      '()
      (cons (f (first L)) (map f (rest L)))))

(define (facs L) (map fac L))

(define (squares L)
   (map (lambda (x) (* x x))
        L
   )
)
```

WHAT DO `filter` AND `sort` DO?

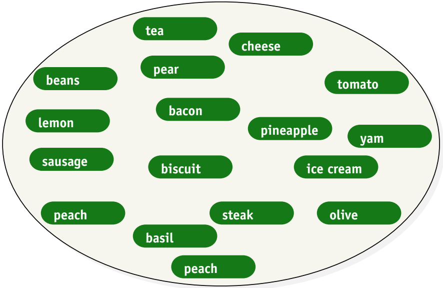
```
(filter odd? '(1 2 3 4 5))    ;; ===> '(1 3 5)

(filter (lambda (n) (> n 3))
        '(1 2 3 4 5))         ;; ===> '(4 5)


(sort '(3 1 2 4 5) <)         ;; ===> '(1 2 3 4 5)

(sort '(3 1 2 4 5) >)         ;; ===> '(5 4 3 2 1)
```

"Use it or lose it" — A general problem-solving strategy

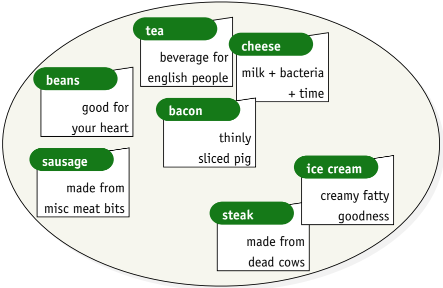
1. Single "it" out
2. Recursively solve the problem **without** "it"
3. Recursively solve the problem **with** "it"
4. Combine (as appropriate)

Two Abstract Interfaces

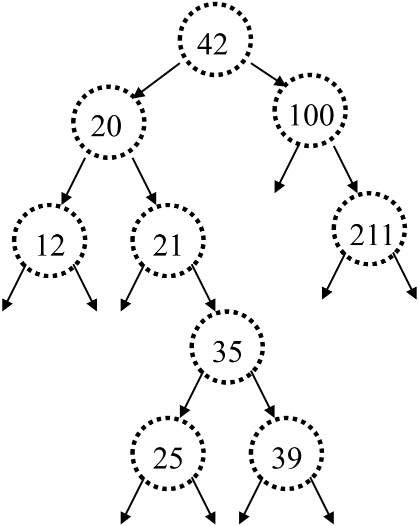
Set: An unordered collection



Map: Associates "keys" with "values"



A Specific Kind of Tree: Binary Search Trees



Identifying Features:

- ✓ Every node has two (possibly empty) subtrees
- ✓ Each node has a "key"
- ✓ The root key is always greater than all nodes in a left subtree
- ✓ The root key is always less than all nodes in a right subtree

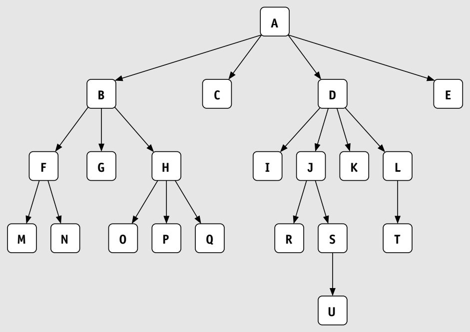
But Racket only has lists...?

Algorithm: Depth-First Search

- ✓ Check the current node
- ✓ (Recursively) search everything reachable from the first child
- ✓ (Recursively) search everything reachable from the second child
- ✓ ...
- ✓ (Recursively) search everything reachable from the last child

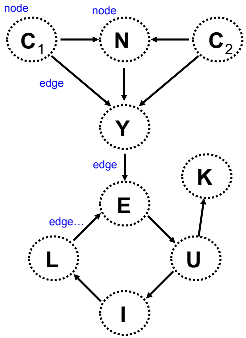
Algorithm: Depth-First Search

- ✓ Check the current node
- ✓ (Recursively) search everything reachable from the first child
- ✓ (Recursively) search everything reachable from the second child
- ✓ ...
- ✓ (Recursively) search everything reachable from the last child



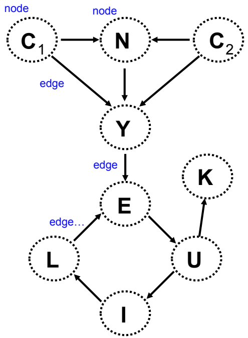
Algorithm: Depth-First Search

- Check the current node
- (Recursively) search everything reachable from the first child
- (Recursively) search everything reachable from the second child
- …
- (Recursively) search everything reachable from the last child



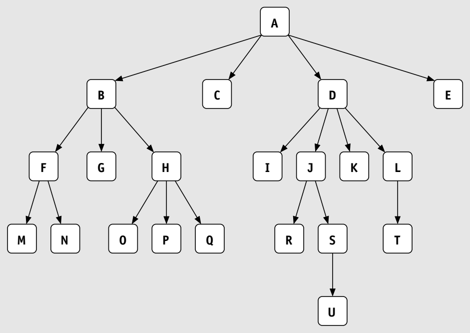
Algorithm: Depth-First Search

- ✓ Check the current node(Optional: Return if you've already searched it)
- ✓ (Recursively) search everything reachable from the first child
- ✓ (Recursively) search everything reachable from the second child
- ✓ …
- ✓ (Recursively) search everything reachable from the last child



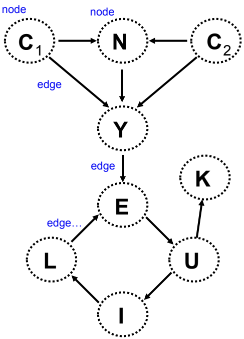
Algorithm: Breadth-First Search

- Check the current node
- Check the children
- Check the children's children
- Check the children's children's children
- ...



Algorithm: Breadth-First Search

- ✓ Check the current node
- ✓ Check the children
- ✓ Check the children's children
- ✓ Check the children's children's children
- ✓ ...



Excerpts from simpsons.pl

```
%% Parent relation
parent(homer, bart).
parent(marge, bart).
parent(homer, lisa).
parent(marge, lisa).
parent(homer, maggie).
parent(marge, maggie).

%% Age relation
age(marge, 35).
age(homer, 38).
age(lisa, 8).
age(maggie, 1).
age(bart, 10).
age(gomer, 41).

%% Female predicate
female(marge).
female(jackie).
female(selma).
female(patty).
female(cher).
female(lisa).

%% Male predicate
male(homer).
male(gomer).
male(gemini).
male(glum).
male(bart).
male(millhouse).
```

```
%% Three rules about families

child(X, Y) :- parent(Y, X).

mother(X, Y) :- female(X), parent(X, Y).

anc(X, Y) :- parent(X, Y).
anc(X, Y) :- parent(Z, Y), anc(X, Z).
```

Using `length` and `member` in Prolog

What should the following Prolog queries do?

```
length([a,b,c,d], 4). ✔
length([a,b,c], 4). ✗
length([a,b,c], N). ✔
length(L, 0). ✔
member(c, [a,b,c,d]). ✔
member(e, [a,b,c,d]). ✗
member(X, [a,b,c,d]). ✔
```

What's a Program?

- ✓ **Python**: A collection of variable definitions, function definitions, and class definitions (and expressions to evaluate).
- ✓ **Racket**: A collection of variable definitions and function definitions (and expressions to evaluate).
- ✓ **Prolog**: A collection of facts and rules.
- ✓ **Java**: A collection of class definitions. (That's it!)

Point.java

```
class Point
{
    ....class contents....
}
```

Objects vs. Classes

Objects

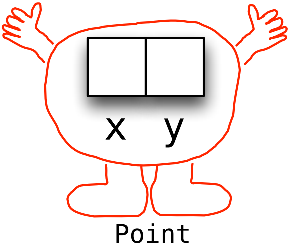
- ✓ Are created at run-time
- ✓ Contain data
- ✓ Have associated code (methods)
- ✓ Have an identity (address in memory)

Classes:

- ✓ Are described at compile-time
- ✓ Provide a "pattern" for describing/creating objects
- ✓ May include other related bits of code and data.

Point OBJECTS

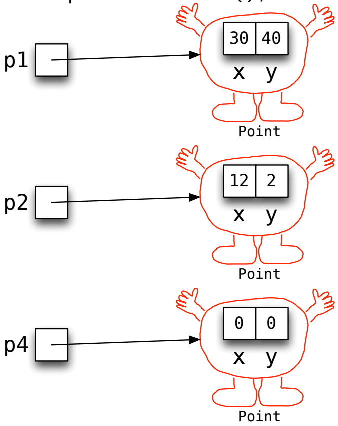
```
class Point extends Object
{
  private double x;
  private double y;
  ...
}
```



Constructors

```
class Point extends Object
{
  ...
  public Point(double x_in,
               double y_in)
  {
    this.x = x_in;
    this.y = y_in;
  }

  public Point()
  {
    this.x = 0.0;
    this.y = 0.0;
  }
  ...
}
```

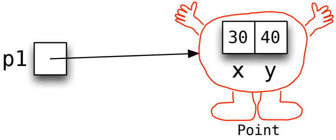
```
Point p1 = new Point(30,40);
Point p2 = new Point(12,2);
...
Point p4 = new Point();
```



METHODS: nudgeBy

```
class Point extends Object
{
  ...
  public void nudgeBy(double delta_x,
                      double delta_y)
  {
    this.x = this.x + delta_x;
    this.y += delta_y;
    return;
  }
  ...
}
```

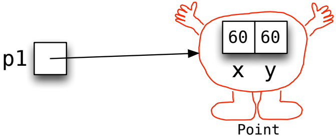
```
Point p1 = new Point(30, 40);

p1.nudgeBy( 30, 20 );
```



Methods: nudgeBy

```
class Point extends Object
{
  ...
  public void nudgeBy(double delta_x,
                      double delta_y)
  {
    this.x = this.x + delta_x;
    this.y += delta_y;
    return;
  }
  ...
}
```

```
Point p1 = new Point(30, 40);

p1.nudgeBy( 30, 20 );
```



Assigning Objects

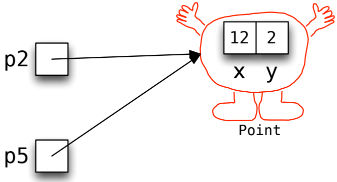
```
Point p2 = new Point(12, 2);
...
Point p5 = p2;
System.out.println("Before nudge:");
System.out.println("p2 is " + p2);
System.out.println("p5 is " + p5);
```

```
p2.nudgeBy(5, 5);

System.out.println("After nudge:");
System.out.println("p2 is " + p2);
System.out.println("p5 is " + p5);
System.out.println("p2 == p5 is " + (p2
```

Assigning Objects

```
Point p2 = new Point(12, 2);
...
Point p5 = p2;
System.out.println("Before nudge:");
System.out.println("p2 is " + p2);
System.out.println("p5 is " + p5);

p2.nudgeBy(5, 5);

System.out.println("After nudge:");
System.out.println("p2 is " + p2);
System.out.println("p5 is " + p5);
System.out.println("p2 == p5 is " + (p2 == p5) );
```

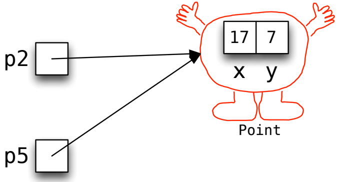
Before nudging:



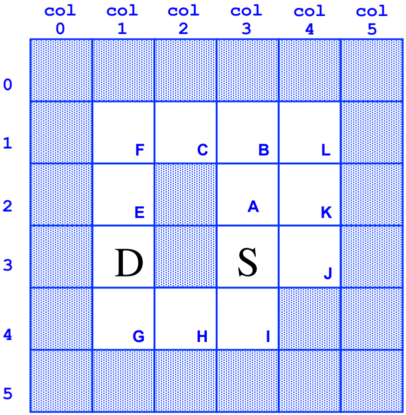
Assigning Objects

```
Point p2 = new Point(12, 2);
...
Point p5 = p2;
System.out.println("Before nudge:");
System.out.println("p2 is " + p2);
System.out.println("p5 is " + p5);

p2.nudgeBy(5, 5);

System.out.println("After nudge:");
System.out.println("p2 is " + p2);
System.out.println("p5 is " + p5);
System.out.println("p2 == p5 is " + (p2 == p5) );
```

After nudging:



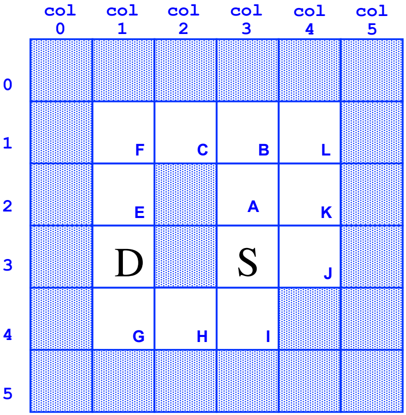
PSEUDOCODE FOR DFS IN A MAZE



'S' = Start Spam Seeking
'D' = Delectible Dinner Destination

```
Create an empty Stack
Mark starting MazeCell as visited
push starting MazeCell onto our Stack

while (the stack's not empty)
{
  current = pop the stack
  for each of current's neighbors
  {
    if (it's not visited or a wall)
    {
     mark it (neighbor) as visited
     set its parent to current MazeCell
     push it onto the stack
    }
  }
}
```

Pseudocode for BFS in a Maze



'S' = Start Spam Seeking
'D' = Delectible Dinner Destination

```
Create an empty Queue
Mark starting MazeCell as visited
enqueue starting MazeCell in our Queue

while (the queue's not empty)
{
  current = dequeue the queue
  for each of current's neighbors
  {
    if (it's not visited or a wall)
    {
     mark it (neighbor) as visited
     set its parent to current MazeCell
     enqueue it in the queue
    }
  }
}
```

Review: StringStack

```
public class StringStack extends Object
{
   private class StackCell {
      private String data;
      private StackCell next; ...
   }

   private StackCell top;

   public StringStack() { ... }
   public void    push(String data) { ... }
   public String  pop()     { if (this.isEmpty()) return null;
                              String topItem = this.top.data;
                              this.top = this.top.next;
                              return topItem; }
   public String  peek()    { ... }
   public boolean isEmpty() { ... }
}
```

Alternative 1: ObjectStack

```
public class ObjectStack extends Object
{
   private class StackCell {
      private Object data;
      private StackCell next; ... }

   private StackCell top;

   public ObjectStack() { ... }

   public void    push(Object data) { ... }
   public Object  pop()     { if (this.isEmpty()) return null;
                              Object topItem = this.top.data;
                              this.top = this.top.next;
                              return topItem; }
   public Object  peek()    { ... }
   public boolean isEmpty() { ... }
}
```

Alternative 2: A Generic Stack

```
public class Stack<T extends Object> extends Object
{
   private class StackCell {
      private T data;
      private StackCell next; ... }

   private StackCell top;

   public Stack() { ... }        // Constructor just called "Stack"
   public void    push(T data)  { ... }
   public T       pop()     { if (this.isEmpty()) return null;
                              T topItem = this.top.data;
                              this.top = this.top.next;
                              return topItem; }
   public T       peek()    { ... }
   public boolean isEmpty() { ... }
}
```

Explain the Difference (1)

```
class Dog {
  private String name;

  public Dog(String dname)
  {
    this.name = dname;
  }

  public void speak()
  {
    System.out.println
     (this.name + ": woof");
  }
}
```

```
class Cat {
  private String name;

  public Cat(String cname)
  {
    this.name = cname;
  }

  public void speak()
  {
    System.out.println
     (this.name + ": meow");
  }
}
```

```
class Animal {
  protected String name;
  public Animal(String n)
  {
     name = n;
  }

  protected void say(String s)
  {
    System.out.println
       (name + ":" + s);
  }
}
```

```
class Dog extends Animal {
  public Dog(String dname)
  {
    super(dname);
  }

  public void speak()
  {
    this.say("woof");
  }
}
```

```
class Cat extends Animal {
  public Cat(String cname)
  {
    super(cname);
  }

  public void speak()
  {
    this.say("meow");
  }
}
```

Explain the Difference (2)

```
class Animal {
  protected String name;
  public Animal(String n)
  {
     this.name = n;
  }

  protected void say(String s)
  {
    System.out.println
     (this.name + ":" + s);
  }
}
```

```
class Dog extends Animal {
  public Dog(String dname)
  {
    super(dname);
  }

  public void speak()
  {
    this.say("woof");
  }
}
```

```
class Cat extends Animal {
  public Cat(String cname)
  {
    super(cname);
  }

  public void speak()
  {
    this.say("meow");
  }
}
```

```
interface Animal {
  public void speak();
}
```

```
class Dog implements Animal {
  private String name;

  public Dog(String dname)
  {
    this.name = dname;
  }

  public void speak()
  {
    System.out.println
     (this.name + ": woof");
  }
}
```

```
class Cat implements Animal {
  private int ears;

  public Cat(int e)
  {
    this.ears = e;
  }

  public void speak()
  {
    System.out.println
     (this.ears + ": meow");
  }
}
```

Important Programming Tip

Inherit only when classes have an "is-a" relationship.

- ✓ a `Kangaroo` is an `Animal`
- ✓ a `Circle` is a `Shape`
- ✓ a `SpamMaze` is a `Maze`

If you're still not sure, think about what Java will let you *do*.

- ✓ If I'm *expecting* an `Animal`, would I *be ok getting* a `Kangaroo`?
- ✓ If I'm *expecting* a `Shape`, would I *be ok getting* a `Circle`?
- ✓ If I'm *expecting* a `Circle`, would I *be ok getting* an `Point`?
- ✓ If I'm *expecting* a `Point`, would I *be ok getting* an `Circle`?

Alternative: Containment

Much more common is the "has-a" relationship.

- ✓ A `Circle` has a central `Point`
- ✓ A `Maze` has a two-dimensional array of `MazeCells`

All we need is a field of the appropriate type.

No inheritance necessary!

Which are OK?

1. `Set<String> myStrings = new HashSet<String>();` ✔
2. `Set<String> myStrings = new TreeSet<String>();` ✔
3. `Set<String> myStrings = new Set<String>();` ✗
4. `HashSet<String> myStrings = new HashSet<String>();` ✔
5. `HashSet<String> myStrings = new TreeSet<String>();` ✗
6. `HashSet<String> myStrings = new Set<String>();` ✗

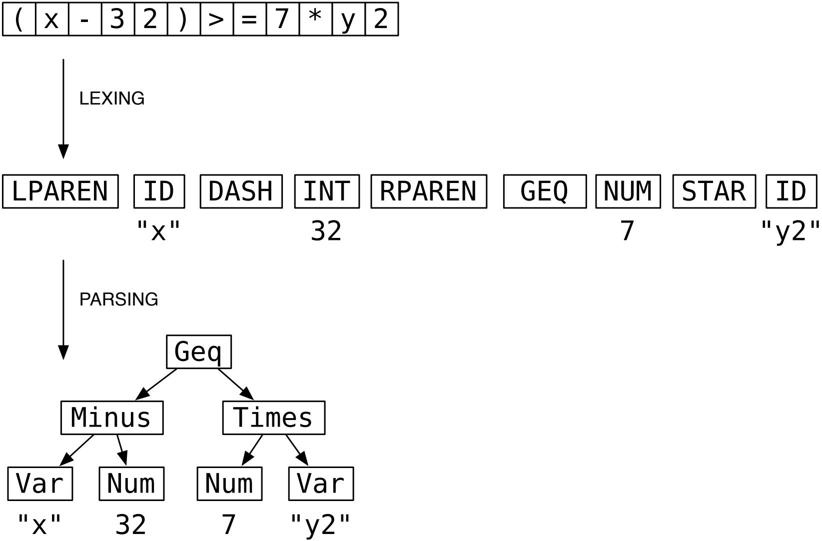
Looping through contents of a set, list, array (!), …

The "new" (Java 5) way: *specialized* **`for`** ("for each") loops:
(Under the hood, Java is creating the iterators for you!)

```
// Print each string in myStrings on its own line

for (String s : myStrings)
{
   System.out.println( s );
}
```

What if we wanted to print each string twice?

Traditional Lexing and Parsing



Regular Expression Ingredients

Regular expressions are a formal way of describing simple patterns.

A regular expression can be:

- ✓ The empty string (sometimes written ϵ or λ)
- ✓ A single character (*e.g.*, $\mathtt{a}$ or 0)
- ✓ Concatenation: $r_1\ r_2$
- ✓ Alternative: $r_1 \mid r_2$
- ✓ Repetition ("Kleene Star"): r_1^*.

In practice we use lots of abbreviations, *e.g.*,

$$\begin{array}{lcl} \texttt{[a-e]} & := & \texttt{(a|b|c|d|e)} \\ r? & := & r \mid \epsilon \\ r^+ & := & r\,r^* \end{array}$$

Application: Regular Expressions and Search

Unix's **egrep** command does line-by-line search for text matching a regular expression.

```
egrep   'hh'        /usr/share/dict/words
egrep   'y.*y'      /usr/share/dict/words
egrep   '(xq|hq)'   /usr/share/dict/words
egrep   '^y.*y$'    /usr/share/dict/words
```

Specifying Syntax via CFGs

A *context-free grammar* is a set of rules for producing a set of strings (a *language*).

$$
\begin{array}{lcl}
S & \rightarrow & V + S \mid V \\
V & \rightarrow & 0 \mid 1 \mid 2 \mid \cdots \mid 9
\end{array}
$$

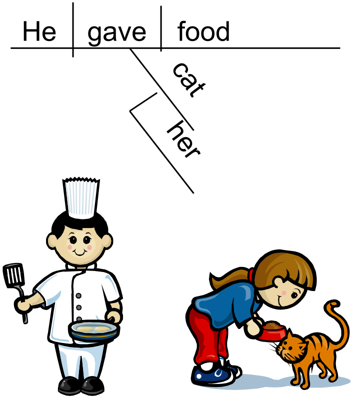
Ingredients:

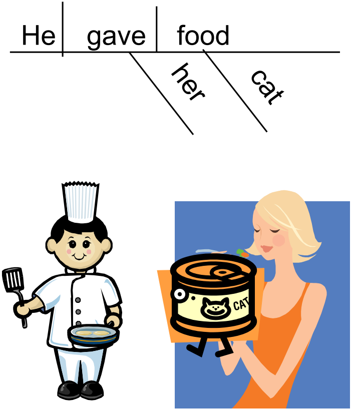
- ✓ Nonterminals: S, V
- ✓ Terminals: $+$, 0, 1, 2, …, 9
- ✓ Production rules: (*see above*)
- ✓ Where to start: S

Show how to produce 4 starting from S.
Show how to produce 4 + 5 starting from S.
What other strings can we produce?

Using Structure to Clarify Meaning

He gave her cat food.





Parse Trees

The *parse tree* of a string makes explicit how a string was produced:

- ✓ Root is the start symbol
- ✓ When we apply a rule, items on the right-hand-side become children

Parse trees for **4** and 4+5?

$$\begin{array}{lcl} S & \rightarrow & V + S \mid V \\ V & \rightarrow & 0 \mid 1 \mid 2 \mid \cdots \mid 9 \end{array}$$

Simple Example of Recursive Descent

```
L -> V
   | V , L
V -> x | y | z
```

```
L():
  # consume tokens matching
  # L in the grammar
  V()

  peek at the next token
  if (it's ","):
     # there's more to this L
     consume the comma
     L()
  else
     # there was only one V.
     return
```

```
V():
  # consume tokens matching
  # V in the grammar
  peek at the next token
  if (it's "x"):
     consume it
  else if (it's "y"):
     consume it
  else if (it's "z"):
     consume it
  else:
    report a parsing error
```

Common Sorting Algorithms

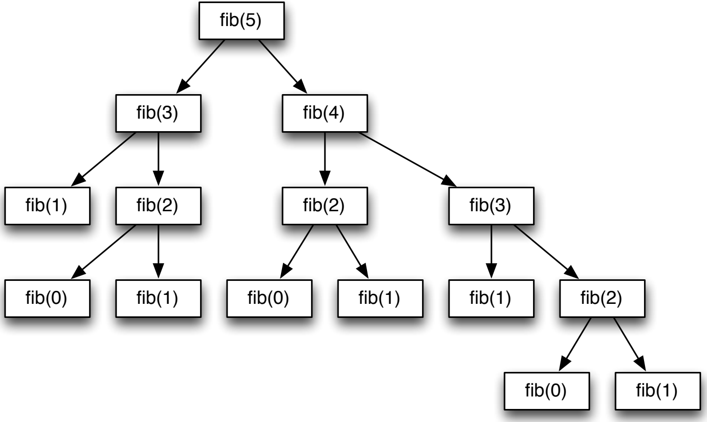
Given an array of n *values:*

Mergesort	$O(n \log n)$	*worst-case*
Quicksort	$O(n \log n)$	*best-case*
	$O(n^2)$	*worst-case*
	$O(n \log n)$	*expected case* (randomly-chosen pivot)
Insertion Sort	$O(n^2)$	*worst-case*
	$O(n)$	*best-case*

CS 70 discusses Heapsort, which is $O(n \log n)$ *worst-case.*

The Problem

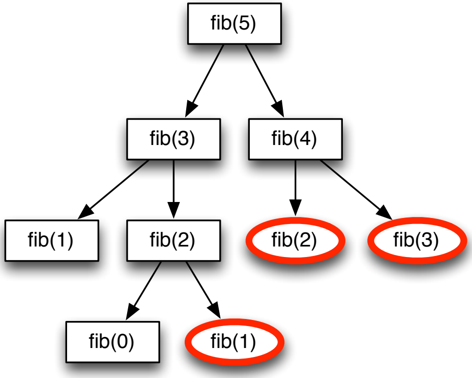
$$\mathrm{fib}(n) = \begin{cases} n & \text{if } n = 0 \text{ or } n = 1 \\ \mathrm{fib}(n-2) + \mathrm{fib}(n-1) & \text{if } n \geq 2 \end{cases}$$



Idea 1: "Memoizing"

Remember all the inputs and output so far
Return precomputed answers for repeated questions.

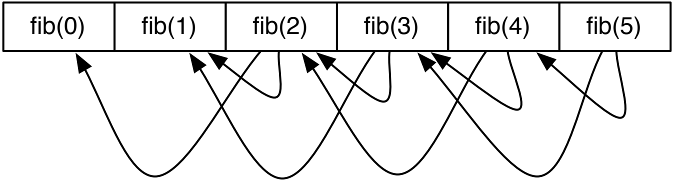
$$\text{fib}(n) = \begin{cases} n & \text{if } n = 0 \text{ or } n = 1 \\ \text{fib}(n-2) + \text{fib}(n-1) & \text{if } n \geq 2 \end{cases}$$



Idea 2: "Dynamic Programming"

Figure out ahead of time which values we'll need.
Compute each exactly once, in a "clever" order.
Ensure that problems are solved *after* their subproblems.

$$\mathrm{fib}(n) = \begin{cases} n & \text{if } n = 0 \text{ or } n = 1 \\ \mathrm{fib}(n-2) + \mathrm{fib}(n-1) & \text{if } n \geq 2 \end{cases}$$



Idealized Computers

We need a precise (mathematical) definition.

Abtract away details that might change.

- ✓ Operating System
- ✓ Processor speed
- ✓ Memory capacity
- ✓ Power source (electricity, natural gas, dilithium, ...)
- ✓ Construction materials (silicon, graphene, legos, ...)
- ✓ Programming language (Java, Racket, Prolog, HMMM, ...)
- ✓ Architecture (single core, multicore, manycore, GPU, VLIW, ...)
- ✓ Data representation (ASCII, Unicode, binary, trinary, ...)

Today's Idealized Computer

State Machine, which

- ✓ A set of possible "configurations" (states)
- ✓ Rules for how the system proceeds from one state to another
  - Depends on current state *and* current input
- ✓ Accept or reject, based on the input this far.

A Jewel of Theoretical Computer Science

The following are equivalent:

1. There is a DFA accepting the set L.
2. There's an NFA accepting L [Rabin and Scott].
3. L can be described by a regular expression [Kleene].

In this case, L is called a **Regular Language**.

More Theorems

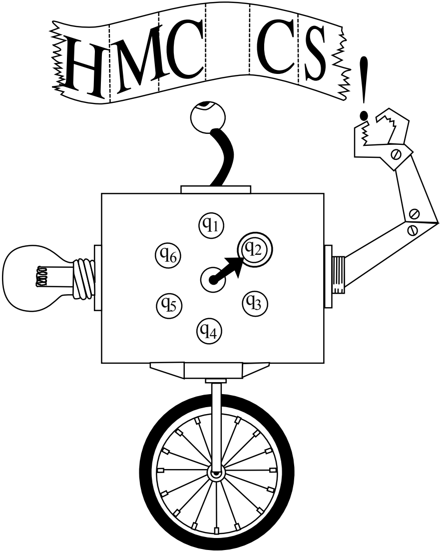
Theorem (Distinguishability Theorem)

If there is a pairwise distinguishable set of N *strings for a language* L*, then a DFA accepting* L *must have* $\geqslant N$ *states.*

Theorem (Nonregular Language Theorem)

If a language L *has an* <u>*infinite*</u> *pairwise distinguishable set of strings, then* L *is not regular.*

Turing Machine: Artist's Conception



Can Turing Machines Solve All Problems?

Church-Turing Thesis: any *plausible* model of computation is no more powerful than a Turing Machine.

- ✓ Proof: by lack of counterexample.

TM Reviews

"Two thumbs way up!"
"The Turing Machine is as good as it gets"

—Alonzo Church and Alan Turing

"A woeful disappointment"
"Missing most of what I'd hope for"

—*Georg Cantor*

Halting is Undecidable

```
def cant(P):
  if HC(P, P):
     while True: pass   # infinite loop
  else:
     return 60
```

Does `cant(cant)` *go* into an infinite loop?

Does `cant(cant)` terminate?

Are there any *practical* reasons to run code
and *give* it its own program as input?

No-Input Halting is Undecidable

```
def HC(P, w):
  """Returns True if P(w) halts; False otherwise"""

  def Q():
    P(w)

  return NIHC(Q)
```

To verify:

if `NIHC` *always returns a correct answer, would* `HC` *always return a correct answer?*

In Conclusion: Computation Is Everywhere

Computers powered by swarms of crabs

16:19 12 April 2012

Computing

Jacob Aron, technology reporter

Yukio-Pegio Gunji of Kobe University in Japan and colleagues realised that when two swarms of crabs collide, they merge and continue in a direction that is the sum of their velocities. This behaviour means the researchers could adapt a previous model of unconventional computing, based on colliding billiard balls, to work with swarms of crabs, with 0s and 1s represented by the absence or presence of a swarm.

They first tried the idea with simulated crab swarms. The OR gate, which simply combines one or two crab swarms into one, worked every time, but the more complicated AND gate, which involves the combined swarm heading down one of three paths, was less reliable.

They then tried the logic gates for real, using swarms of 40 crabs. The crab swarms were placed at the entrances of the logic gates and encouraged to move by a looming shadow that fooled them into thinking a predatory bird was overhead. The results closely matched the simulation, suggesting that crab-powered computers could indeed be possible.

http://www.newscientist.com/blogs/onepercent/2012/04/researchers-build-crab-powered.html