Recursion
Basic and Complex Recursion

Brad Miller    David Ranum

12/19/2005
Outline

1. Objectives
2. What Is Recursion?
   - Calculating the Sum of a List of Numbers
   - The Three Laws of Recursion
   - Converting an Integer to a String in Any Base
3. Stack Frames: Implementing Recursion
4. Complex Recursive Problems
   - Tower of Hanoi
   - Sierpinski Triangle
   - Cryptography and Modular Arithmetic
5. Summary
Objectives

To understand that complex problems that may otherwise be difficult to solve may have a simple recursive solution.

To learn how to formulate programs recursively.

To understand and apply the three laws of recursion.

To understand recursion as a form of iteration.

To implement the recursive formulation of a problem.

To understand how recursion is implemented by a computer system.
Outline

1. Objectives

2. What Is Recursion?
   - Calculating the Sum of a List of Numbers
   - The Three Laws of Recursion
   - Converting an Integer to a String in Any Base

3. Stack Frames: Implementing Recursion

4. Complex Recursive Problems
   - Tower of Hanoi
   - Sierpinski Triangle
   - Cryptography and Modular Arithmetic

5. Summary
The Iterative Sum Function

```python
1  def listsum(l):
2      sum = 0
3      for i in l:
4          sum = sum + i
5      return sum
```
Recursive listSum

```
1   def listsum(l):
2       if len(l) == 1:
3           return l[0]
4       else:
5           return l[0] + listsum(l[1:])
```
Series of Recursive Calls Adding a List of Numbers

\[
\begin{align*}
\text{sum}(1,3,5,7,9) &= 1 + \\
\text{sum}(3,5,7,9) &= 3 + \\
\text{sum}(5,7,9) &= 5 + \\
\text{sum}(7,9) &= 7 + \\
\text{sum}(9) &= 9
\end{align*}
\]
Series of Recursive Returns from Adding a List of Numbers

Calculating the Sum of a List of Numbers

The Three Laws of Recursion

Converting an Integer to a String in Any Base
Outline

1. Objectives

2. What Is Recursion?
   - Calculating the Sum of a List of Numbers
   - The Three Laws of Recursion
   - Converting an Integer to a String in Any Base

3. Stack Frames: Implementing Recursion

4. Complex Recursive Problems
   - Tower of Hanoi
   - Sierpinski Triangle
   - Cryptography and Modular Arithmetic

5. Summary
A recursive algorithm must have a **base case**.

2. A recursive algorithm must change its state and move toward the base case.

3. A recursive algorithm must call itself, recursively.
Outline

1. Objectives
2. What Is Recursion?
   - Calculating the Sum of a List of Numbers
   - The Three Laws of Recursion
   - Converting an Integer to a String in Any Base
3. Stack Frames: Implementing Recursion
4. Complex Recursive Problems
   - Tower of Hanoi
   - Sierpinski Triangle
   - Cryptography and Modular Arithmetic
5. Summary
1. Reduce the original number to a series of single-digit numbers.
2. Convert the single digit-number to a string using a lookup.
3. Concatenate the single-digit strings together to form the final result.
Converting an Integer to a String in Base 10

- `toStr(769)`
  - `769 / 10` → '9'
- `toStr(76)`
  - `76 / 10` → '6'
- `toStr(7)`
  - `7 < 10` → '7'

Remainder

```plaintext
769 / 10 = '9'
76 / 10 = '6'
7 < 10 = '7'
```
Converting an Integer to a String in Base 2–16

```python
1  convertString = "0123456789ABCDEF"
2
3  def toStr(n, base):
4      if n < base:
5          return convertString[n]
6      else:
7          return toStr(n / base, base) + convertString[n%base]
```
Converting the Number 10 to its Base 2 String Representation

```
toStr(10) -> 10 / 2 + '0'
  |    |
  v    v
toStr(5) -> 5 / 2 + '1'
  |    |   |
  v    v  |
  toStr(2) -> 2 / 2 + '0'
     |    |   |
     v    v |
     toStr(1) -> 1 < 2 + '1'
```

Remainder
Pushing the Strings onto a Stack

```python
def toStr(n, base):
    if n < base:
        rStack.push(convertString[n])
    else:
        rStack.push(convertString[n % base])
        toStr(n / base, base)
```

1. convertString = "0123456789ABCDEF"
2. rStack = Stack()

Recursion
Strings Placed on the Stack During Conversion

Diagram of stack frames during conversion:

- '1'
- '0'
- '1'
- '0'

Recursion
Call Stack Generated from `toStr(10, 2)`

```
1

toStr(2, 2)
  n = 5
  base = 2
  toStr(2/2, 2) + convertString[2%2]

toStr(5, 2)
  n = 5
  base = 2
  toStr(5/2, 2) + convertString[5%2]

toStr(10, 2)
  n = 10
  base = 2
  toStr(10/2, 2) + convertString[10%2]
```
Outline

1. Objectives

2. What Is Recursion?
   - Calculating the Sum of a List of Numbers
   - The Three Laws of Recursion
   - Converting an Integer to a String in Any Base

3. Stack Frames: Implementing Recursion

4. Complex Recursive Problems
   - Tower of Hanoi
   - Sierpinski Triangle
   - Cryptography and Modular Arithmetic

5. Summary
An Example Arrangement of Disks for the Tower of Hanoi
Tower of Hanoi

1. Move a tower of height-1 to an intermediate pole, using the final pole.
2. Move the remaining disk to the final pole.
3. Move the tower of height-1 from the intermediate pole to the final pole using the original pole.
Python Code for the Tower of Hanoi

```
1 def moveTower(height, fromPole, toPole, withPole):
2     if height >= 1:
3         moveTower(height-1, fromPole, withPole, toPole)
4         moveDisk(fromPole, toPole)
5         moveTower(height-1, withPole, toPole, fromPole)
```
Python Code to Move One Disk

```python
def moveDisk(fp, tp):
    print "moving disk from %d to %d\n" % (fp, tp)
```
Outline

1. Objectives
2. What Is Recursion?
   - Calculating the Sum of a List of Numbers
   - The Three Laws of Recursion
   - Converting an Integer to a String in Any Base
3. Stack Frames: Implementing Recursion
4. Complex Recursive Problems
   - Tower of Hanoi
   - Sierpinski Triangle
   - Cryptography and Modular Arithmetic
5. Summary
The Sierpinski Triangle
Code for the Sierpinski Triangle

```python
def sierpinskiT(points, level, win):
    colormap = ['blue','red','green','white',
                'yellow','violet','orange']
    p = Polygon(points)
    p.setFill(colormap[level])
    p.draw(win)
    if level > 0:
        sierpinskiT([points[0],getMid(points[0],points[1]),
                      getMid(points[0],points[2])],level-1,win)
        sierpinskiT([points[1],getMid(points[0],points[1]),
                      getMid(points[1],points[2])],level-1,win)
        sierpinskiT([points[2],getMid(points[2],points[1]),
                      getMid(points[0],points[2])],level-1,win)
```

Recursion
def getMid(p1, p2):
    return Point(((p1.getX()+p2.getX()) / 2.0),
                  ((p1.getY()+p2.getY()) / 2.0))

if __name__ == '__main__':
    win = GraphWin('st', 500, 500)
    win.setCoords(20, -10, 80, 50)
    myPoints = [Point(25, 0), Point(50, 43.3), Point(75, 0)]
    sierpinskiT(myPoints, 6, win)
Building a Sierpinski Triangle
Outline

1. Objectives

2. What Is Recursion?
   - Calculating the Sum of a List of Numbers
   - The Three Laws of Recursion
   - Converting an Integer to a String in Any Base

3. Stack Frames: Implementing Recursion

4. Complex Recursive Problems
   - Tower of Hanoi
   - Sierpinski Triangle
   - Cryptography and Modular Arithmetic

5. Summary
A Simple Modular Encryption Function

```python
1
def encrypt(m):
2    s = 'abcdefghijklmnopqrstuvwxyz'
3    n = ''
4    for i in m:
5        j = (s.find(i)+13)%26
6        n = n + s[j]
7    return n
```
Decryption Using a Simple Key

```python
1   def decrypt(m, k):
2       s = 'abcdefghijklmnopqrstuvwxyz'
3       n = ''
4       for i in m:
5           j = (s.find(i)26-k)%26
6           n = n + s[j]
7       return n
```
1. If \( a \equiv b \pmod{n} \) then \( \forall c, a + c \equiv b + c \pmod{n} \).
2. If \( a \equiv b \pmod{n} \) then \( \forall c, ac \equiv bc \pmod{n} \).
3. If \( a \equiv b \pmod{n} \) then \( \forall p, p > 0, a^p \equiv b^p \pmod{n} \).
1. Initialize `result` to 1.
2. Repeat `n` times:
   1. Multiply `result` by `x`.
   2. Apply modulo operation to `result`. 
Recursive Definition for $x^n \pmod{p}$

```python
1 def modexp(x, n, p):
2     if n == 0:
3         return 1
4     t = (x*x)%p
5     tmp = modexp(t, n/2, p)
6     if n%2 != 0:
7         tmp = (tmp * x) % p
8     return tmp
```
Euclid’s Algorithm for GCD

```python
1 def gcd(a, b):
2     if b == 0:
3         return a
4     elif a < b:
5         return gcd(b, a)
6     else:
7         return gcd(a-b, b)
```
An Improved Euclid’s Algorithm

```python
def gcd(a, b):
    if b == 0:
        return a
    else:
        return gcd(b, a % b)
```

Recursion
Extended GCD

```python
def ext_gcd(x, y):
    if y == 0:
        return (x, 1, 0)
    else:
        (d, a, b) = ext_gcd(y, x % y)
        return (d, b, a - (x / y) * b)
```
RSA KeyGen Algorithm

def RSAgenKeys(p, q):
    n = p * q
    pqminus = (p-1) * (q-1)
    e = int(random.random() * n)
    while gcd(pqminus, e) != 1:
        e = int(random.random() * n)
    d, a, b = ext_gcd(pqminus, e)
    if b < 0:
        d = pqminus + b
    else:
        d = b
    return ((e, d, n))
RSA Encrypt Algorithm

```python
def RSAencrypt(m, e, n):
    ndigits = len(str(n))
    chunkSize = ndigits - 1
    chunks = toChunks(m, chunkSize)
    encList = []
    for messChunk in chunks:
        print(messChunk)
        c = modexp(messChunk, e, n)
        encList.append(c)
    return encList
```
RSA Decrypt Algorithm

```python
1  def RSAdecrypt(clist, d, n):
2      rList = []
3      for c in clist:
4          m = modexp(c, d, n)
5          rList.append(m)
6      return rList
```
Recursion Summary

- All recursive algorithms must have a base case.
- A recursive algorithm must change its state and make progress toward the base case.
- A recursive algorithm must call itself (recursively).
- Recursion can take the place of iteration in some cases.
- Recursive algorithms often map very naturally to a formal expression of the problem you are trying to solve.
- Recursion is not always the answer. Sometimes a recursive solution may be more computationally expensive than an alternative algorithm.