Recursion

Basic and Complex Recursion

Brad Miller David Ranum

12/19/2005

Outline

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

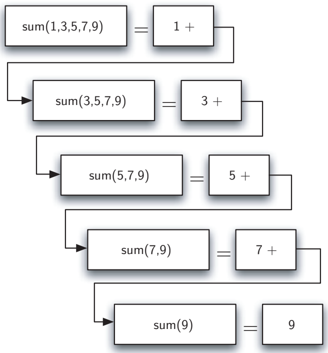
The Iterative Sum Function

```
def listsum(l):
    sum = 0
    for i in l:
        sum = sum + i
    return sum
```

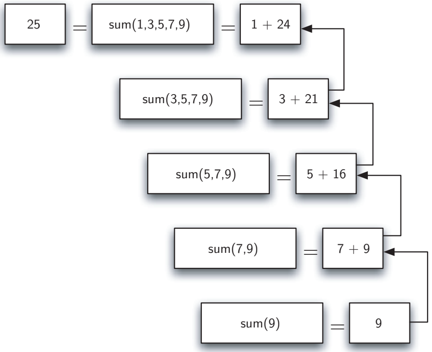
Recursive listSum

```
def listsum(l):
    if len(l) == 1:
        return l[0]
    else:
        return l[0] + listsum(l[1:])
```

Series of Recursive Calls Adding a List of Numbers



Series of Recursive Returns from Adding a List of Numbers



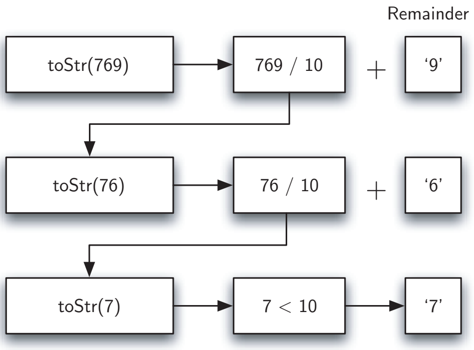
Outline

1. 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. 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

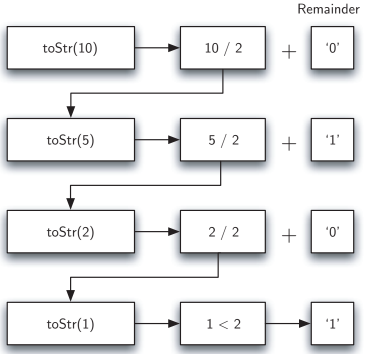


Converting an Integer to a String in Base 2–16

```
convertString = "0123456789ABCDEF"

def toStr(n,base):
    if n < base:
        return convertString[n]
    else:
        return toStr(n / base,base) + convertString[n%base]
```

Converting the Number 10 to its Base 2 String Representation

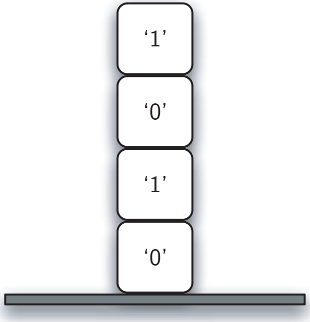


Pushing the Strings onto a Stack

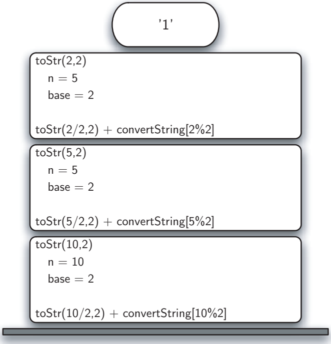
```
convertString = "0123456789ABCDEF"
rStack = Stack()

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

Strings Placed on the Stack During Conversion

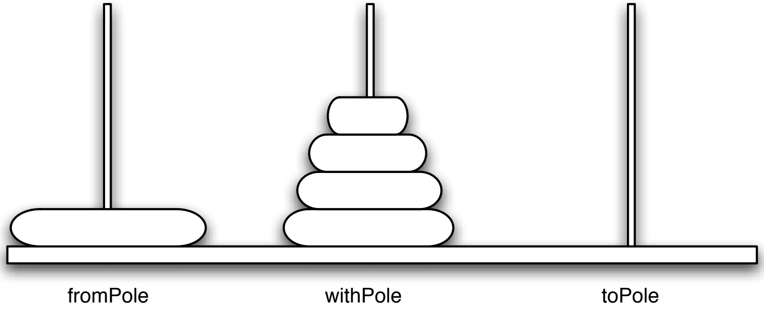


Call Stack Generated from `toStr(10,2)`



Outline

An Example Arrangement of Disks for the 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

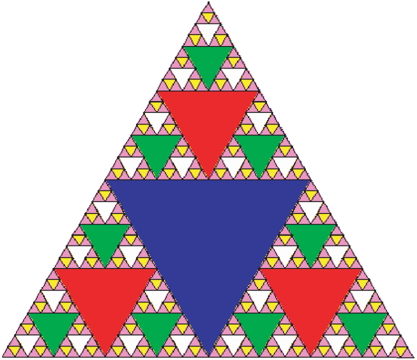
```
def moveTower(height,fromPole, toPole, withPole):
 if height >= 1:
moveTower(height-1,fromPole,withPole,toPole)
moveDisk(fromPole,toPole)
moveTower(height-1,withPole,toPole,fromPole)
```

Python Code to Move One Disk

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

Outline

The Sierpinski Triangle



Code for the Sierpinski Triangle I

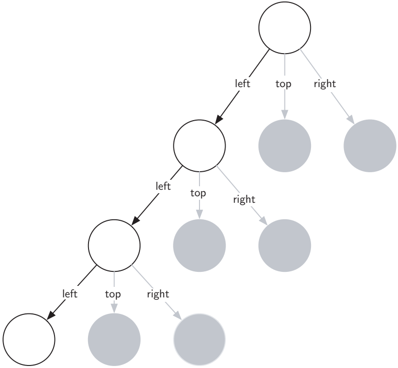
```
def sierpinskiT(points,level,win):
    colormap = ['blue','red','green','white',
                'yellow','violet','orange']
    p = Polygon(points)
    p.setFill(colormap[level])
    p.draw(win)
    if level > 0:
        sierpinskiT([points[0],getMid(points[0],points[1]),
                getMid(points[0],points[2])],level-1,win)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

Code for the Sierpinski Triangle II

```
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

A Simple Modular Encryption Function

```

def encrypt(m):
    s = 'abcdefghijklmnopqrstuvwxyz'
    n = ''
    for i in m:
        j = (s.find(i)+13)%26
        n = n + s[j]
    return n
```

Decryption Using a Simple Key

```
def decrypt(m,k):
    s = 'abcdefghijklmnopqrstuvwxyz'
    n = ''
    for i in m:
        j = (s.find(i)26-k)%26
        n = n + s[j]
    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 $x^n \pmod{p}$

```
def modexp(x,n,p):
    if n == 0:
        return 1
    t = (x*x)%p
    tmp = modexp(t,n/2,p)
    if n%2 != 0:
        tmp = (tmp * x) % p
    return tmp
```

Euclid's Algorithm for GCD

```
def gcd(a,b):
    if b == 0:
        return a
    elif a < b:
        return gcd(b,a)
    else:
        return gcd(a-b,b)
```

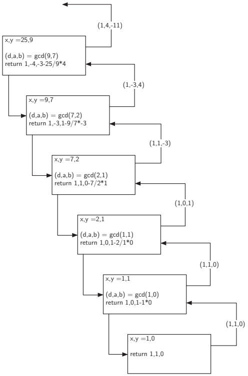
An Improved Euclid's Algorithm

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

Extended GCD

```
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)
```

Call Tree for Extended GCD Algorithm



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

```
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

```
def RSAdecrypt(clist,d,n):
    rList = []
    for c in clist:
        m = modexp(c,d,n)
        rList.append(m)
    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.