Iterative algorithms

- Looping constructs (e.g. while or for loops) lead naturally to **iterative** algorithms
- Can conceptualize as capturing computation in a set of “state variables” which update on each iteration through the loop

```
def iterMul(a, b):
    result = 0
    while b > 0:
        result += a
        b -= 1
    return result
```

Iterative multiplication by successive additions

- Imagine we want to perform multiplication by successive additions:
  - To multiply a by b, add a to itself b times
- State variables:
  - i – iteration number; starts at b
  - result – current value of computation; starts at 0
- Update rules
  - i ← i -1; stop when 0
  - result ← result + a

Recursive version

- An alternative is to think of this computation as:

$$a * b = \underbrace{a + a + \ldots + a}_{\text{b copies}}$$

$$= a + \underbrace{a + \ldots + a}_{\text{b-1 copies}}$$

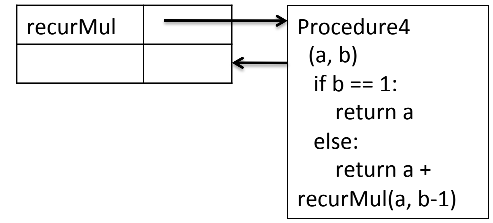
$$= a + a * (b – 1)$$

Recursion

- This is an instance of a **recursive** algorithm
  - Reduce a problem to a simpler (or smaller) version of the same problem, plus some simple computations
    - **Recursive step**
  - Keep reducing until reach a simple case that can be solved directly
    - **Base case**
- a * b = a; if b = 1 (Base case)
- a * b = a + a * (b-1); otherwise (Recursive case)

Let's try it out

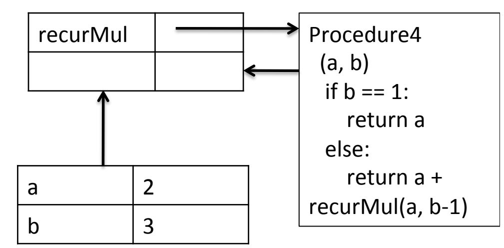
```
def recurMul(a, b):
    if b == 1:
        return a
    else:
        return a +
  recurMul(a, b-1)
```




```
def recurMul(a, b):
    if b == 1:
        return a
    else:
        return a + recurMul(a, b-1)
```

Let's try it out

```
def recurMul(a, b):
    if b == 1:
        return a
    else:
        return a +
  recurMul(a, b-1)
```

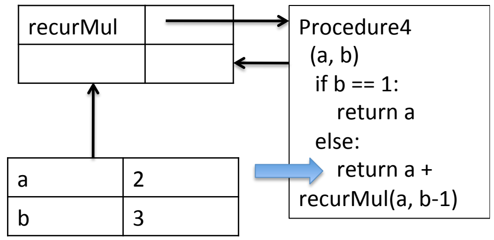


recurMul(2, 3)

Let's try it out

```
def recurMul(a, b):
    if b == 1:
        return a
    else:
        return a +
  recurMul(a, b-1)
```

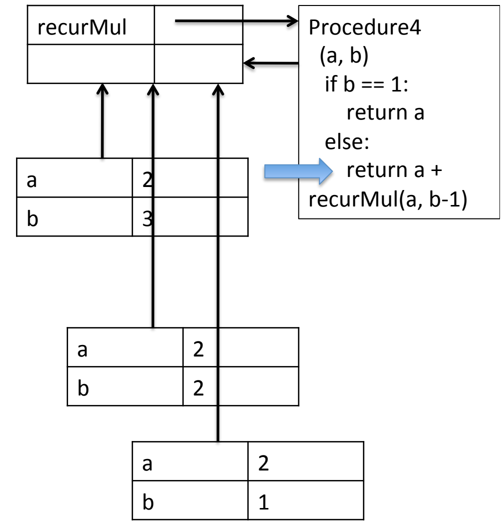
recurMul(2, 3)



Let's try it out

```
def recurMul(a, b):
    if b == 1:
        return a
    else:
        return a +
  recurMul(a, b-1)
```

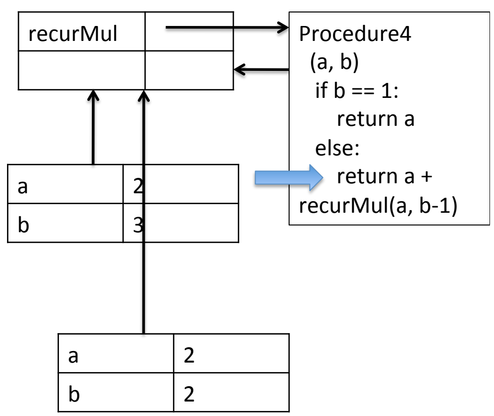
recurMul(2, 3)



Let's try it out

```
def recurMul(a, b):
    if b == 1:
        return a
    else:
        return a +
  recurMul(a, b-1)
```

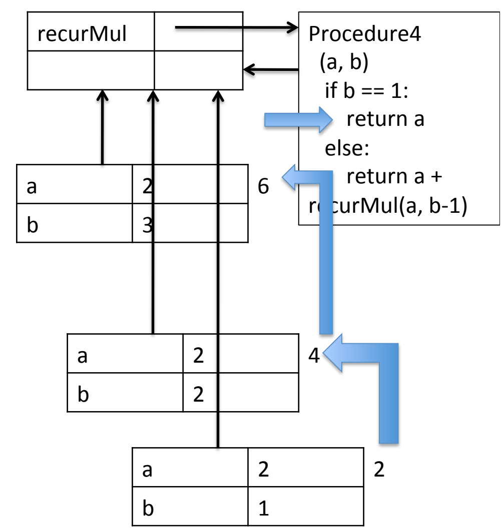
recurMul(2, 3)



Let's try it out

```
def recurMul(a, b):
    if b == 1:
        return a
    else:
        return a +
  recurMul(a, b-1)
```

recurMul(2, 3)



Some observations

- Each recursive call to a function creates its own environment, with local scoping of variables
- Bindings for variable in each frame distinct, and not changed by recursive call
- Flow of control will pass back to earlier frame once function call returns value

Mathematical induction

- To prove a statement indexed on integers is true for all values of n:
  - Prove it is true when n is smallest value (e.g. n = 0 or n = 1)
  - Then prove that if it is true for an arbitrary value of n, one can show that it must be true for n+1

Inductive reasoning

- How do we know that our recursive code will work?
- iterMul terminates because b is initially positive, and decrease by 1 each time around loop; thus must eventually become less than 1
- recurMul called with b = 1 has no recursive call and stops
- recurMul called with b > 1 makes a recursive call with a smaller version of b; must eventually reach call with b = 1

Example

- $0 + 1 + 2 + 3 + \ldots + n = (n(n+1))/2$
- Proof
  - If n = 0, then LHS is 0 and RHS is $0*1/2 = 0$, so true
  - Assume true for some k, then need to show that
    - $0 + 1 + 2 + \ldots + k + (k+1) = ((k+1)(k+2))/2$
    - LHS is $k(k+1)/2 + (k+1)$ by assumption that property holds for problem of size k
    - This becomes, by algebra, $((k+1)(k+2))/2$
  - Hence expression holds for all n >= 0

What does this have to do with code?

- Same logic applies

```
def recurMul(a, b):
    if b == 1:
        return a
    else:
        return a + recurMul(a, b-1)
```

- Base case, we can show that recurMul must return correct answer
- For recursive case, we can assume that recurMul correctly returns an answer for problems of size smaller than b, then by the addition step, it must also return a correct answer for problem of size b
- Thus by induction, code correctly returns answer

```
def factI(n):
   """assumes that n is
 an int > 0
      returns n!"""
   res = 1
   while n > 1:
       res = res * n
       n -= 1
   return res
```

```
def factR(n):
   """assumes that n is
 an int > 0
      returns n!"""
   if n == 1:
       return n
   return n*factR(n-1)
```

The "classic" recursive problem

- Factorial

n! = n * (n-1) * … * 1

$$= \begin{cases} n * (n-1)! & \text{if } n > 1 \\ 1 & \text{if } n = 1 \end{cases}$$

Towers of Hanoi

- The story:
  - 3 tall spikes
  - Stack of 64 different sized discs – start on one spike
  - Need to move stack to second spike (at which point universe ends)
  - Can only move one disc at a time, and a larger disc can never cover up a small disc

Towers of Hanoi

- Having seen a set of examples of different sized stacks, how would you write a program to print out the right set of moves?
- Think recursively!
  - Solve a smaller problem
  - Solve a basic problem
  - Solve a smaller problem

```
def printMove(fr, to):
    print('move from ' + str(fr) + ' to ' + str(to))

def Towers(n, fr, to, spare):
    if n == 1:
        printMove(fr, to)
    else:
        Towers(n-1, fr, spare, to)
        Towers(1, fr, to, spare)
        Towers(n-1, spare, to, fr)
```

Recursion with multiple base cases

- Fibonacci numbers
  - Leonardo of Pisa (aka Fibonacci) modeled the following challenge
    - Newborn pair of rabbits (one female, one male) are put in a pen
    - Rabbits mate at age of one month
    - Rabbits have a one month gestation period
    - Assume rabbits never die, that female always produces one new pair (one male, one female) every month from its second month on.
    - How many female rabbits are there at the end of one year?

Fibonacci

- After one month (call it 0) – 1 female
- After second month – still 1 female (now pregnant)
- After third month – two females, one pregnant, one not
- In general, females(n) = females(n-1) + females(n-2)
  - Every female alive at month n-2 will produce one female in month n;
  - These can be added those alive in month n-1 to get total alive in month n

Month	Females
0	1
1	1
2	2
3	3
4	5
5	8
6	13

Fibonacci

- Base cases:
  – Females(0) = 1
  – Females(1) = 1
- Recursive case
  – Females(n) = Females(n-1) + Females(n-2)

```
def fib(x):
    """assumes x an int >= 0
       returns Fibonacci of x"""
    assert type(x) == int and x >= 0
    if x == 0 or x == 1:
        return 1
    else:
        return fib(x-1) + fib(x-2)
```


Recursion on non-numerics

- How could we check whether a string of characters is a palindrome, i.e., reads the same forwards and backwards
  – "Able was I ere I saw Elba" – attributed to Napolean
  – "Are we not drawn onward, we few, drawn onward to new era?"

How to we solve this recursive?

- First, convert the string to just characters, by stripping out punctuation, and converting upper case to lower case
- Then
  – Base case: a string of length 0 or 1 is a palindrome
  – Recursive case:
    - If first character matches last character, then is a palindrome if middle section is a palindrome

Example

- ‘Able was I ere I saw Elba’ → ‘ablewasiereisawleba’
- isPalindrome(‘ablewasiereisawleba’) is same as
  - ‘a’ == ‘a’ and isPalindrome(‘blewasiereisawleb’)

```
def isPalindrome(s):

    def toChars(s):
        s = s.lower()
        ans = ''
        for c in s:
            if c in 'abcdefghijklmnopqrstuvwxyz':
                ans = ans + c
        return ans

    def isPal(s):
        if len(s) <= 1:
            return True
        else:
            return s[0] == s[-1] and isPal(s[1:-1])

    return isPal(toChars(s))
```

Divide and conquer

- This is an example of a “divide and conquer” algorithm
  - Solve a hard problem by breaking it into a set of sub-problems such that:
    - Sub-problems are easier to solve than the original
    - Solutions of the sub-problems can be combined to solve the original

Global variables

- Suppose we wanted to count the number of times fib calls itself recursively
- Can do this using a global variable
- So far, all functions communicate with their environment through their parameters and return values
- But, (though a bit dangerous), can declare a variable to be global – means name is defined at the outermost scope of the program, rather than scope of function in which appears

Example

```
def fibMetered(x):
    global numCalls
    numCalls += 1
    if x == 0 or x == 1:
        return 1
    else:
        return fibMetered(x-1) + fibMetered(x-2)


def testFib(n):
    for i in range(n+1):
        global numCalls
        numCalls = 0
        print('fib of ' + str(i) + ' = ' + str(fibMetered(i)))
        print('fib called ' + str(numCalls) + ' times')
```

Global variables

- Use with care!!
- Destroy locality of code
- Since can be modified or read in a wide range of places, can be easy to break locality and introduce bugs!!