CSI33 Data Structures

Department of Mathematics and Computer Science
Bronx Community College

Chapter 6: Recursion

- Recursive Definitions
- Simple Recursive Examples
- Analyzing Recursion
- In-Class Work

Recursive Definitions

A Function Can Call Itself

- A recursive definition of a function is one which makes a function call to the function being defined.
- The function call is then a recursive function call.
- A definition is circular if it leads to an infinite sequence of function calls.
- To prevent this:
  - *the function must call itself with a parameter smaller than the one it is using*.
  - the function must test for when the parameter has reached the minimum size (the base case(s)): this must be handled without a recursive call.

The Call Stack

The *function call stack* can handle recursive functions easily. There is no reason why a function can't push an activation record onto the call stack with variables for the current function while calling that same function. The earlier version of that function will resume when the recursive call is completed.

When the base case is finally met, there will be no further recursive calls, and no further activation records will be pushed onto the stack.

Without a base case, the stack would overflow, producing a run-time error.

Recursive Definitions

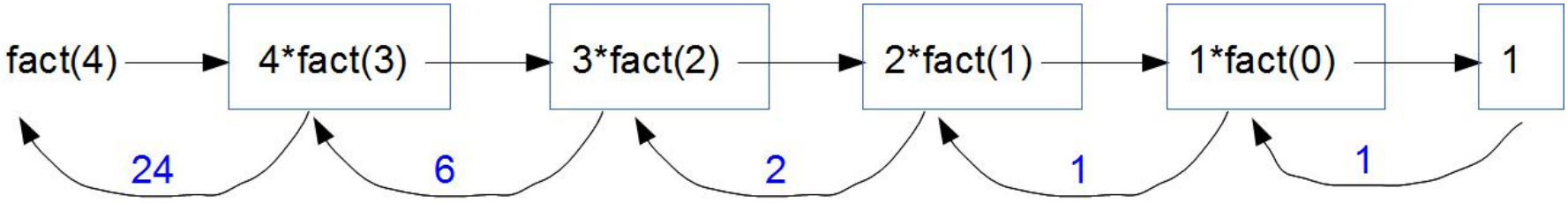
The Factorial Function

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n(n-1)! & \text{otherwise} \end{cases}$$

The Factorial Function - using Python

```
def fact(n):
    if n == 0:
        return 1
    else:
        return n * fact(n - 1)
```

Recursive Definitions



The Factorial Function - using Python

```
def fact(n):
    if n == 0:
        return 1
    else:
        return n * fact(n - 1)
```

See fact.py

String Reversal

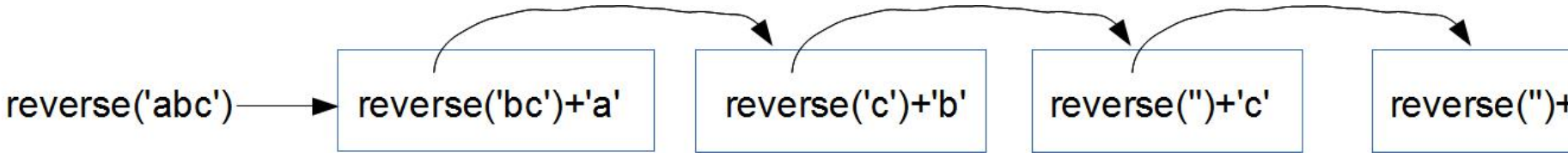
Circular Definition

```
def reverse(s):
   return reverse(s[1:])  + s[0]
```

String Reversal

Circular Definition

```
def reverse(s):
    return reverse(s[1:])  + s[0]
```



String Reversal

Definition with Base Case

```
def reverse(s):
   if s == "":
      return s
   else:
      return reverse(s[1:])  + s[0]
```

See reverse.py

Anagrams

An anagram of a word is another word spelled using the same letters but rearranged. Rearrangements are also called permutations. For example: TORSO is an anagram for ROOST.

A recursive strategy to produce all anagrams of a given word is:

- remove the first letter from the word.
- for all anagrams of the smaller word, insert it in all possible positions.

How many permutations of the letters in TORSO is there?

Anagrams

An anagram of a word is another word spelled using the same letters but rearranged. Rearrangements are also called permutations. For example: TORSO is an anagram for ROOST.

A recursive strategy to produce all anagrams of a given word is:

- remove the first letter from the word.
- for all anagrams of the smaller word, insert it in all possible positions.

How many permutations of the letters in TORSO is there?
$5! = 120$

Anagrams Using Recursion

```
def anagrams(s):
    if s == "":
        return [s]
    else:
        ans = []
        for w in anagrams(s[1:]):
            for pos in range(len(w)+1):
                ans.append(w[:pos]+s[0]+w[pos:1])
        return ans
```

See anagrams.py

Fast Exponentiation

Naive Iteration is $\Theta(n)$

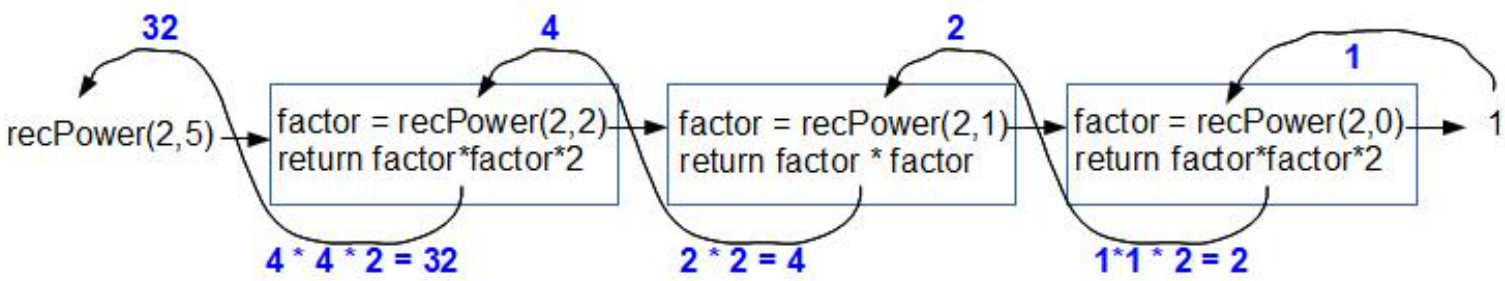
```
# power.py
def loopPower(a, n):
    ans = 1
    for i in range(n):
        ans = ans * a
    return ans
```

Fast Exponentiation

Divide and Conquer Recursion is $\Theta(\log\ n)$

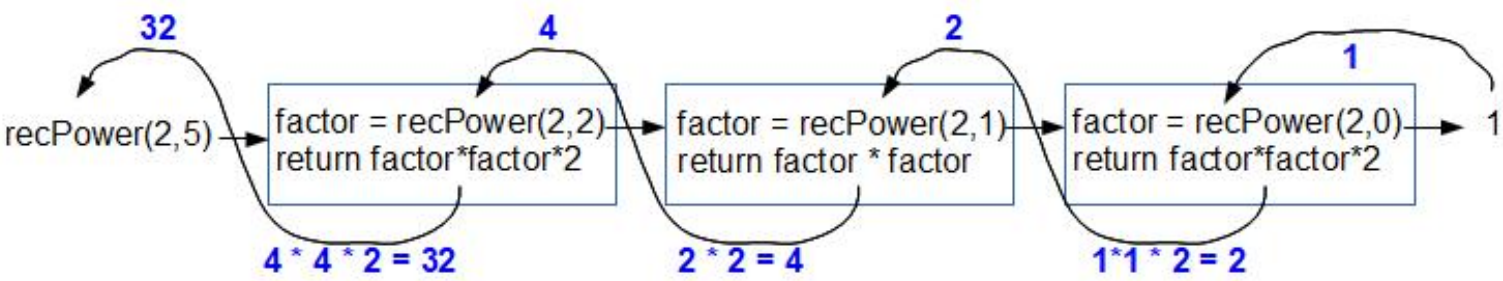
```
# power.py
def recPower(a, n):
    if n == 0:
        return 1
    else:
        factor = recPower(a, n // 2)
        if n % 2 == 0:
            return factor * factor
        else:
            return factor * factor * a
```

Fast Exponentiation




```
# power.py
def recPower(a, n):
    if n == 0:
        return 1
    else:
        factor = recPower(a, n // 2)
        if n % 2 == 0:
            return factor * factor
        else:
            return factor * factor * a
```

Fast Exponentiation



Let's compare the number of multiplications performed while using recursive definition of power function and naive definition of power function:

naive (running time $\Theta(n)$): 4 multiplications
recursive (running time $\Theta(\log n)$): 5 multiplications

If we try 2^{10}, then
naive: 9 multiplications
recursive: 6 multiplications

Binary Search

Iteration

```
def search(items, target):
    low = 0
    high = len(items) - 1
    while low <= high:
        mid = (low + high) // 2
        item = nums[mid]
        if target == item:
            return mid
        elif target < item:
            high = mid - 1
        else:
            low = mid + 1
    return -1
```

Binary Search

Pseudocode Using Recursion

```
Algorithm:  binary search
-- search for x in nums[low]...nums[high]
   if low > high
      x is not in nums
   mid = (low + high) // 2
   if x== nums[mid]:
      x is at mid position
   elif x < nums[mid]
      binary search for x in nums[low]...nums[mid-1]
   else
      binary search for x in nums[mid+1]...nums[high]
```

Binary Search

Python Code Using Recursion

```
def search(items, target):
    return recBinSearch(target, items, 0, len(items)-1)
def recBinSearch(x, nums, low, high):
    if low > high:
        return -1
    mid = (low + high) // 2
    item = nums[mid]
    if x == item:
        return mid
    elsif x < item:
        return recBinSearch(x, nums, low, mid-1)
    else:
        return recBinSearch(x, nums, mid+1, high)
```

Comparison With Iterative (Looping) Algorithms

- Any iterative algorithm can be transformed into a recursive one.
- Different strategies lead to different running times. (The recursive power example is more efficient than the naive loop version.)
- To measure efficiency, you must count **recursive calls** and **the depth of the call stack**.
- You must also consider the **size of the data parameters** that are passed in recursive calls.

The Fibonacci Sequence

The Fibonacci Sequence is obtained by beginning with the pair of numbers $1, 1$ and continuing indefinitely by adding the last two numbers to give the next number in the sequence, giving 1, 1, 2, 3, 5, 8, 13 and so on.

The Fibonacci Sequence

The nth Fibonacci Number: Loop Version

```
def loopFib(n):
    curr = 1
    prev = 1
    for i in range(n - 2):
        curr, prev = curr + prev, curr
    return curr
```

Analysis

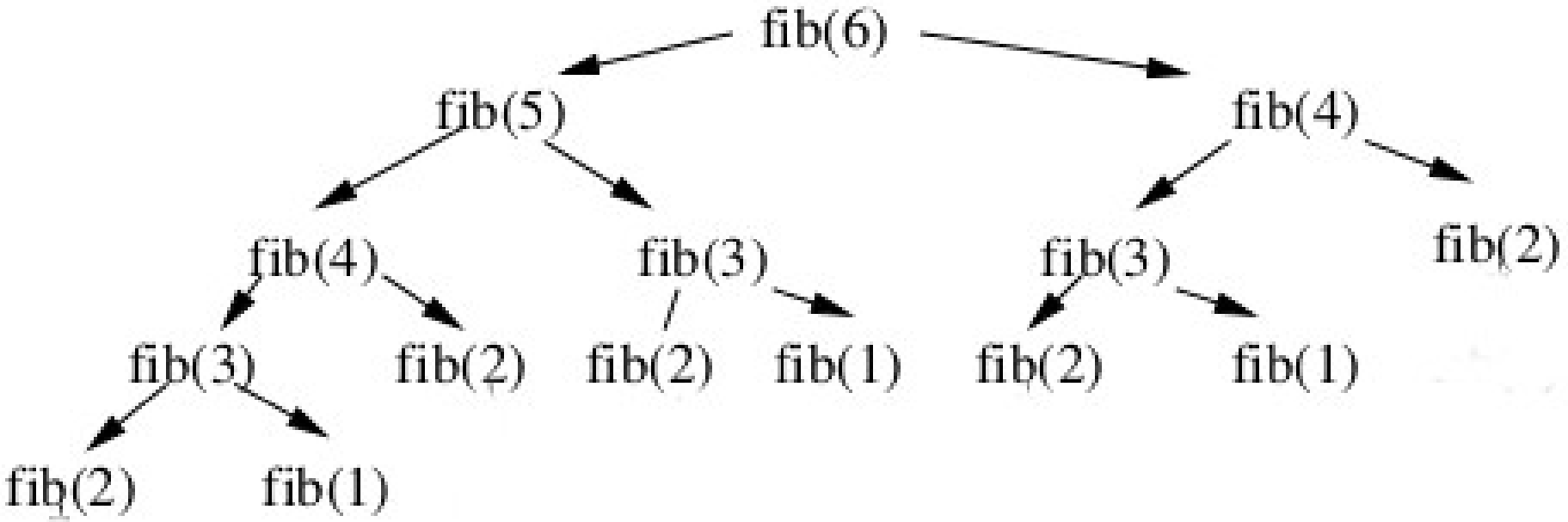
To calculate `fib(n)` requires $n - 2$ iterations of the for loop, so the running time is $\Theta(n)$.

The Fibonacci Sequence

The nth Fibonacci Number: Recursive Version

```
def recFib(n):
    if n < 3:
        return 1
    else:
        return recFib(n - 1) + recFib(n - 2)
```

The Fibonacci Sequence



The Fibonacci Sequence

Analysis

To calculate `fib(6)` is very wasteful:

- `fib(4)` is calculated 2 times
- `fib(3)` is calculated 3 times
- `fib(2)` is calculated 5 times
- `fib(1)` is calculated 3 times

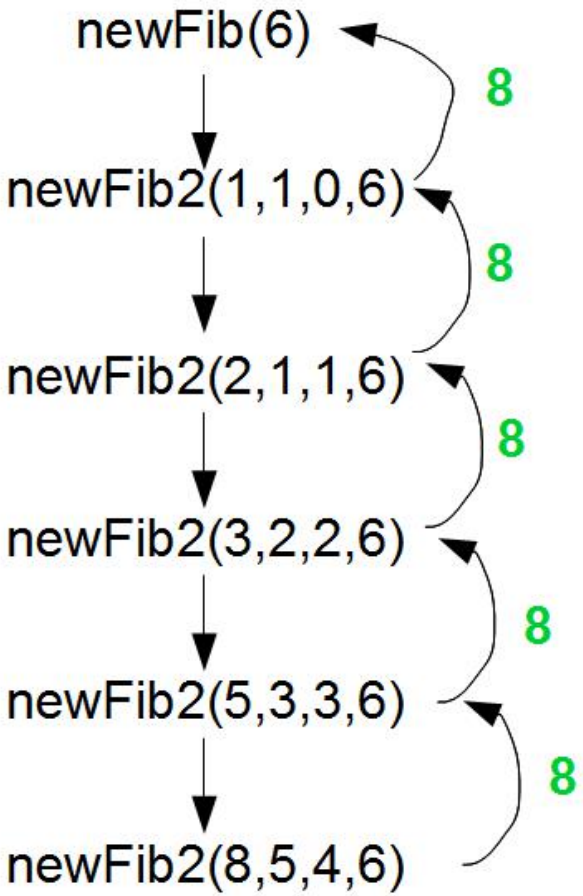
  To calculate `fib(n)` requires $fib(n) - 1$ steps, so the running time is $\Theta(fib(n))$, which is $\Theta(2^n)$, or exponential in n.

The nth Fibonacci Number: Improved Recursive Version

```
def newFib(n):
    return newFib2(1, 1, 0, n)
def newFib2(curr, prev, i, n):
    if i == n - 2:
        return curr
    else:
        return newFib2(curr + prev,curr, i + 1, n)
```

You can see that this definition won't work if a user wants to get the first Fibonacci number, i.e. 1, but it works perfectly well for all other cases. Can it be fixed? (class work)

The Fibonacci Sequence



Analysis

To calculate `fib(n)` now requires $n - 2$ recursive calls, so the running time is $\Theta(n)$, which is a big improvement.

How To Make An Iterative Function Recursive

- Write a function that calls a helper function with parameters for all local variables and parameters from the loop version.
- Pass the initial values from the loop version in this function call.
- The helper function will be recursive:
- The base case will be the negation of the loop condition.
- The recursive call will change the parameters to match one iteration of the loop version.

In-Class Work

1. Show pictorial representation of the call `recPower(3,7)`.
2. Figure out exactly how many multiplications does `recPower(3,7)` do.
3. Write and test a recursive function `Maximum` to find the largest number in a list.
   Hint: the maximum is the larger of the first item and maximum of all the other items.
4. Fix the `newFib` function to make it work when `newFib(1)` is called.