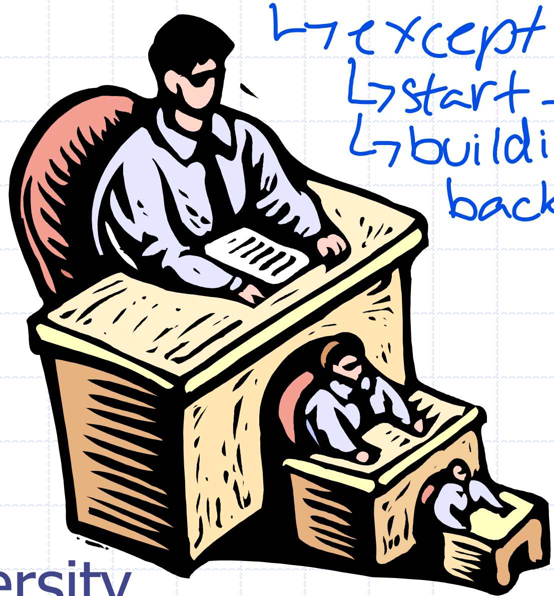
CH4

fractal → pattern that repeat
↳ different zoom levels

loop
↳ depend on one another
↳ except inner
↳ start - base case
↳ building backwards

Recursion



Sareh Taebi
COP3410 – Florida Atlantic University

Introduction

- Recursion: When one function calls itself during execution.
- Fractals are recursive in nature
- Russian dolls have recursive pattern in art.
- In computing, recursion provides an elegant and powerful alternative for performing repetitive tasks.

3 factorial: 3x2x1 → 3 factorial ← find permutations?
3!
6 permutations
ABC CAB BAC
BCA CBA ACB
3 options 2 options 1 option
$\boxed{3} \times \boxed{2} \times \boxed{1}$

The Recursion Pattern

- The factorial function:
- n! = 1· 2· 3· ··· · (n-1)· n = n . (n-1)!
- 0! = 1   1! = 1
- Recursive definition:

$$f(n) = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot f(n-1) & else \end{cases}$$

- As a Python method:

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

def fac(n): //return factorial
// n! = n×(n-1)×...×2×1
O(n)
fact = 0
for i in range(2,n)
O(1) fact = i × fact
return fact
//fact = 1
// fact = 1×2
// fact = 2×3
//fact = 6×4

Content of a Recursive Method

if :
known → base case
else :
recursion

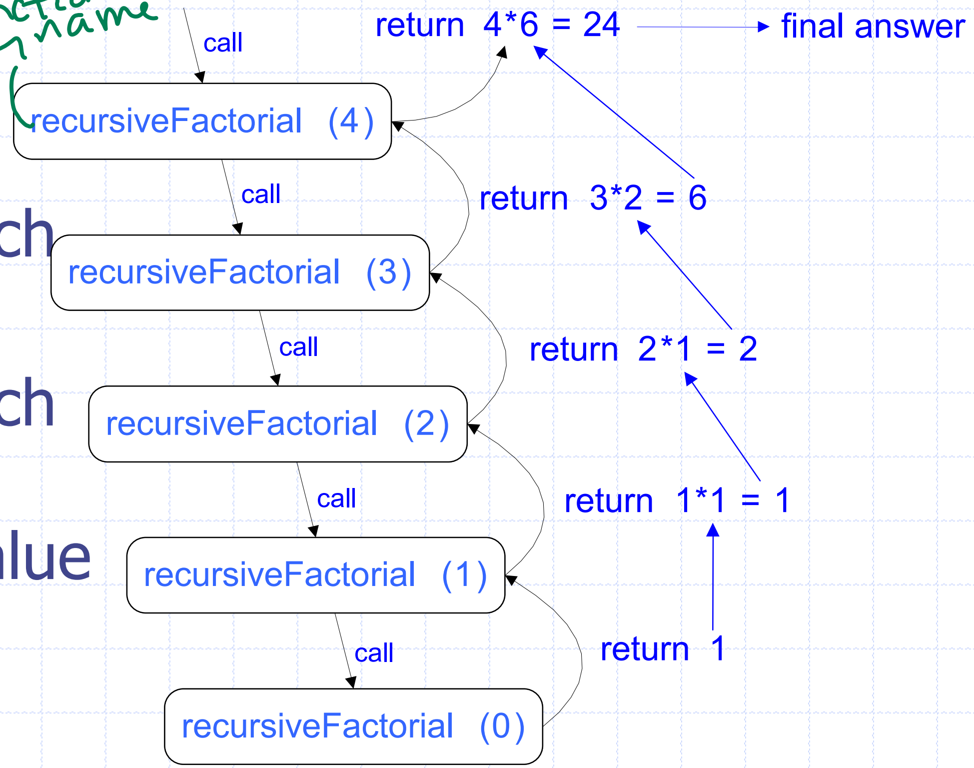
- Base case(s)
  - Values of the input variables for which we perform no recursive calls are called base cases (there should be at least one base case).
    ↳ known answers
  - Every possible chain of recursive calls must eventually reach a base case.
- Recursive calls
  - Calls to the current method.
  - Each recursive call should be defined so that it makes progress towards a base case.

Visualizing Recursion

O(n)
↳ linear time

- Recursion trace
  - A box for each recursive call
    calling itself
  - An arrow from each caller to callee
  - An arrow from each callee to caller showing return value
- Example

function name



Analyzing the Recursive Factorial Function

- Analyze inside every activation frame:
  - A total of n+1 activations from n! down to 0! : o(n)
  - A constant number of operations in each activation : o(1)
- factorial(n) is o(n)

O(n) ↳find max

when dealing w/data
↳sort first
→ sorted(n)
↳python function

Binary Search

- Search for an integer, target, in an ordered list.

```
def binary_search(data, target, low, high):
  """Return True if target is found in indicated portion of a Python list.

  The search only considers the portion from data[low] to data[high] inclusive.
  """
  if low > high:
    return False                                   # interval is empty; no match
  else:
    mid = (low + high) // 2
    if target == data[mid]:                        # found a match
      return True
    elif target < data[mid]:
      # recur on the portion left of the middle
      return binary_search(data, target, low, mid − 1)
    else:
      # recur on the portion right of the middle
      return binary_search(data, target, mid + 1, high)
```

find errors

base

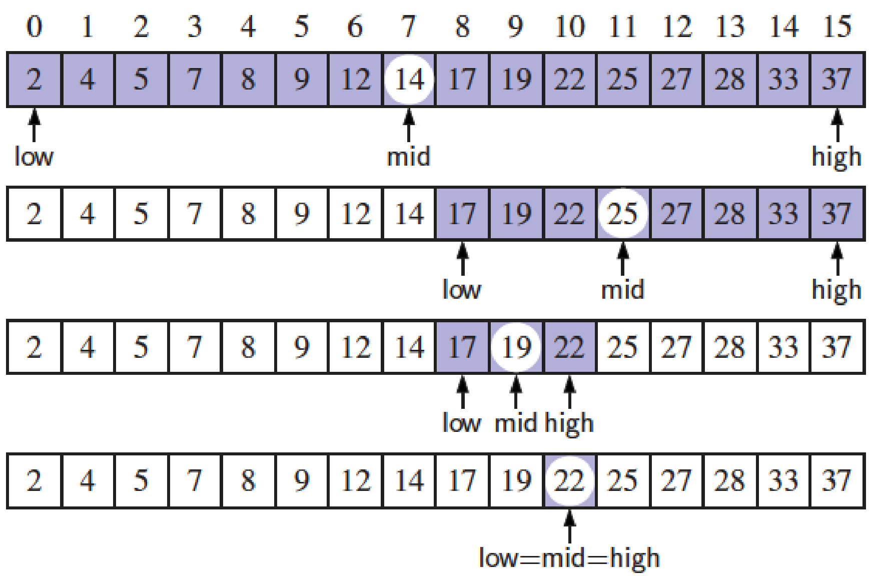
recursion

recursion

becomes low

Visualizing Binary Search

- We consider three cases:
  - If the target equals data[mid], then we have found the target.
  - If target < data[mid], then we recur on the first half of the sequence.
  - If target > data[mid], then we recur on the second half of the sequence.



Analyzing Binary Search

- Runs in **O(log n)** time.
  - The remaining portion of the list is of size high – low + 1.
  - After one comparison, this becomes one of the following:

$$(\mathrm{mid}-1)-\mathrm{low}+1=\left\lfloor\frac{\mathrm{low}+\mathrm{high}}{2}\right\rfloor-\mathrm{low}\leq\frac{\mathrm{high}-\mathrm{low}+1}{2}$$

$$\mathrm{high}-(\mathrm{mid}+1)+1=\mathrm{high}-\left\lfloor\frac{\mathrm{low}+\mathrm{high}}{2}\right\rfloor\leq\frac{\mathrm{high}-\mathrm{low}+1}{2}.$$

  - Thus, each recursive call divides the search region in half; hence, there can be at most *log n* levels.