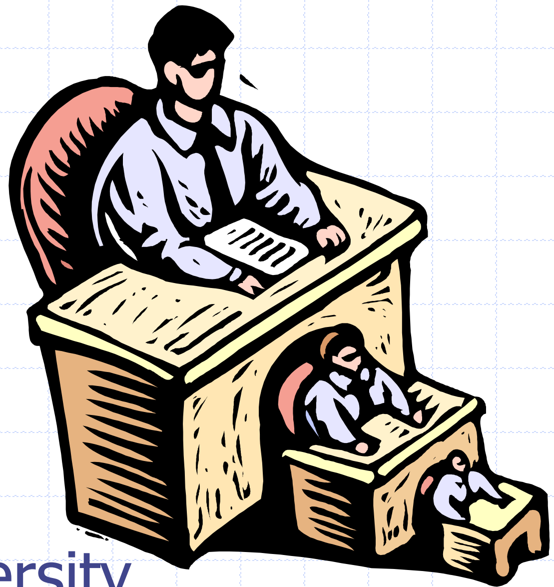
fractal repeats itself at different zoom levels
like repeating a computation inside itself through a function

```
def factorial(n):
    return factorial(n-1) n    // recursion- calling inside itself
```

Recursion II

To practice:

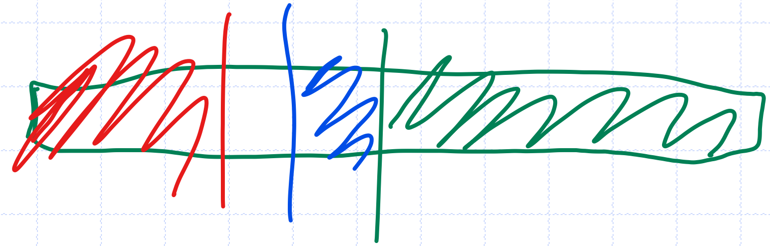
- ↳ factorial recursion
- ↳ computing total /array
- ↳ binary search recursion
- ↳ reversing array order



Sareh Taebi
COP3410 – Florida Atlantic University

- search list with less complexity
- list has to be sorted
  - sorted(data)- low complexity 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)
```

↳ base case

can also return location

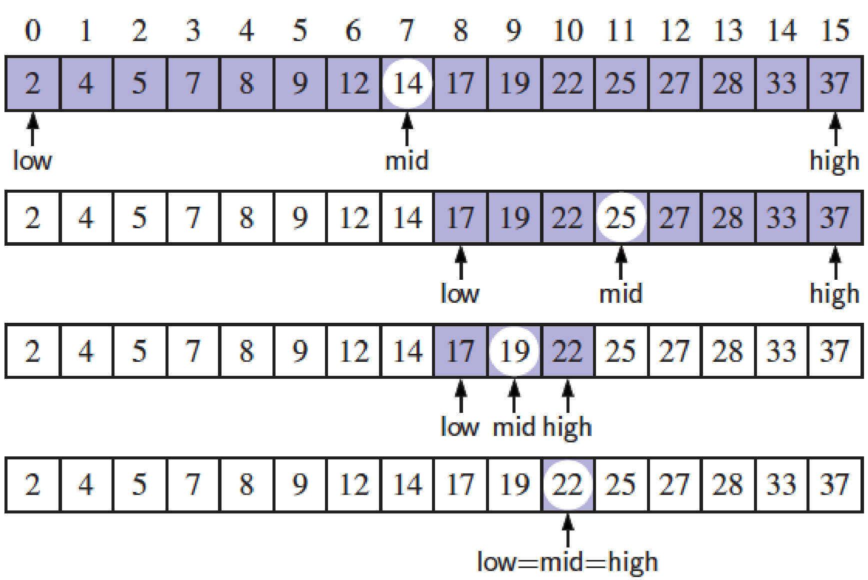
↳ recursion

↳ change lower bond

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.

size = 16
targ = 22
divide by 2
compare target to mid
divide by 2
compare target to new mid



w/o binary search
- have to compare each to target
- sequential search is O(n)

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.

Linear Recursion

↳ only calls inside function once
↳ called again separately
↳ different if / else statement

- Test for base cases
  - Begin by testing for a set of base cases (there should be at least one).
  - Every possible chain of recursive calls must eventually reach a base case, and the handling of each base case should not use recursion.
- Recur once
  - Perform a single recursive call
  - This step may have a test that decides which of several possible recursive calls to make, but it should ultimately make just one of these calls
  - Define each possible recursive call so that it makes progress towards a base case.

if she asks for recursive function
if/else ↳ 3 or 4 branches

[1, 2, 3, 4, 5]

Example of Linear Recursion

Algorithm LinearSum(*A*, *n*):

Input:

A integer array *A* and an integer *n* = 1, such that *A* has at least *n* elements

Output:

The sum of the first *n* integers in *A*

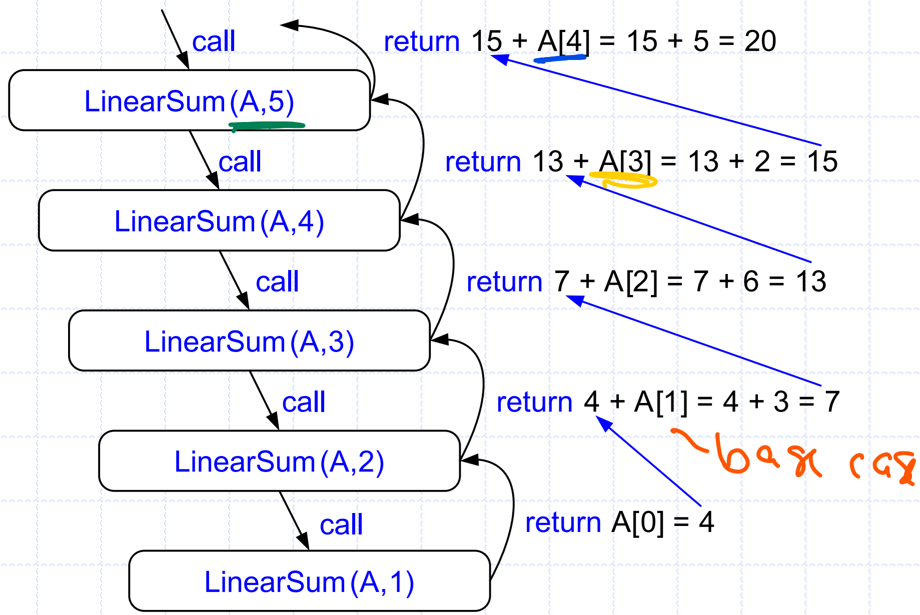
if *n* = 1 **then**

  return *A*[0]

else

  return LinearSum(*A*, *n* - 1) + *A*[*n* - 1]

Example recursion trace:



Reversing an Array

Algorithm ReverseArray(A, i, j):

Input: An array A and nonnegative integer indices i and j

Output: The reversal of the elements in A starting at index i and ending at j

Pseudocode

if $i < j$ **then**

  Swap $A[i]$ and $A[j]$

  ReverseArray(A, $i + 1$, $j - 1$)

return

Base case
↳ if i==j → if 1st & last are the same

return array

swap(a,b):
temp = a
a = b
b = temp

Defining Arguments for Recursion

- In creating recursive methods, it is important to define the methods in ways that facilitate recursion.
- This sometimes requires we define additional paramaters that are passed to the method.
- For example, we defined the array reversal method as ReverseArray(*A*, *i*, *j*), not ReverseArray(*A*).
- Python version:

```
def reverse(S, start, stop):
  """Reverse elements in implicit slice S[start:stop]."""
  if start < stop − 1:                              # if at least 2 elements:
    S[start], S[stop−1] = S[stop−1], S[start]       # swap first and last
    reverse(S, start+1, stop−1)                     # recur on rest
```

swap
↳ a,b = b,a

$x^n = x^{n-1} \cdot x \to x^{n-2} \cdot x \to x^{n-3} \cdot x$ } Recursion

n: input size

can start w/ multiple base cases & fix

Base cases: definite

↳ $n=0 \to x^0 = 1$

↳ $n=1 \to x^1 = x$

Computing Powers

- The power function, $p(x,n)=x^n$, can be defined recursively:

$$\overbrace{p(x,n)}^{x^n} = \begin{cases} 1 & \text{if } n = 0 \\ x \cdot p(x, n-1) & \text{else} \end{cases}$$

↳power

- This leads to an power function that runs in **O(n)** time (for we make n recursive calls).
- We can do better than this, however.

$x^4 = x^{(4/2)^2}$ $\quad x^5 = x^{(5/2)^2} + 1$

$\underbrace{x^n \to x^{(n/2)}}$

↳ cases odd or even

↳ decrease complexity

Recursive Squaring

- We can derive a more efficient linearly recursive algorithm by using repeated squaring:

$$p(x,n) = \begin{cases} 1 & \text{if } x = 0 \\ x \cdot p(x,(n-1)/2)^2 & \text{if } n > 0 \text{ is odd} \\ p(x,n/2)^2 & \text{if } n > 0 \text{ is even} \end{cases}$$

- For example,

$O(\log n)$

$2^4 = 2^{(4/2)2} = (2^{4/2})^2 = (2^2)^2 = 4^2 = 16$

$2^5 = 2^{1+(4/2)2} = 2(2^{4/2})^2 = 2(2^2)^2 = 2(4^2) = 32$

$2^6 = 2^{(6/2)2} = (2^{6/2})^2 = (2^3)^2 = 8^2 = 64$

$2^7 = 2^{1+(6/2)2} = 2(2^{6/2})^2 = 2(2^3)^2 = 2(8^2) = 128.$

Recursive Squaring Method

Algorithm Power(x, n):

Input: A number x and integer $n = 0$

Output: The value x^n

if $n = 0$ **then**

$x^0 = 1$

return 1

if n is odd **then**

$x^3 = x^{(n/2)^2+1}$

$y = \text{Power}(x, (n-1)/2)$

return $x \cdot y \cdot y$

$\left(\frac{n-1}{2}\right)^2 = \frac{n^2+1-2n}{4}$

else

$y = \text{Power}(x, n/2)$

$x^4 = x^{(4/2)^2}$

return $y \cdot y$

Analysis

Algorithm Power(x, n):
Input: A number x and integer $n = 0$
Output: The value x^n
if $n = 0$ **then**
  return 1
if n is odd **then**
  y = Power(x, (n - 1)/ 2)
  return *x* ' *y* ' *y*
else
  y = Power(x, n/ 2)
  return *y* ' *y*

Each time we make a recursive call we halve the value of n; hence, we make log n recursive calls. That is, this method runs in O(log n) time.

It is important that we use a variable twice here rather than calling the method twice.