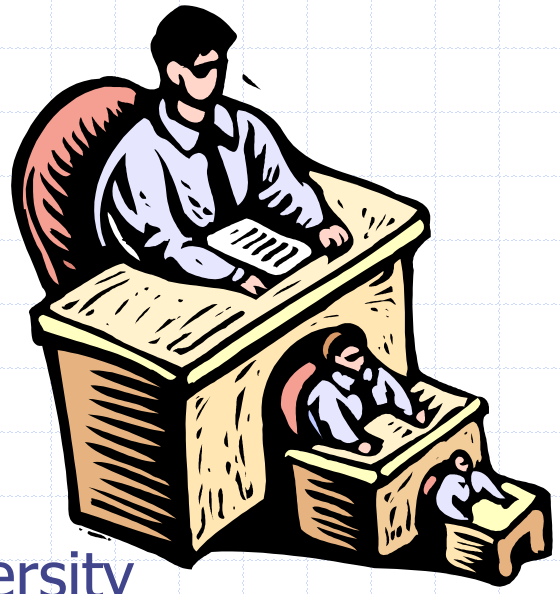
fractal repeats itself at different zoom levels
like repeating a computation inside itself through a function

```
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.

```
1  def binary_search(data, target, low, high):
2    """Return True if target is found in indicated portion of a Python list.
3
4    The search only considers the portion from data[low] to data[high] inclusive.
5    """
6    if low > high:
7      return False                          # interval is empty; no match
8    else:
9      mid = (low + high) // 2
10     if target == data[mid]:               # found a match
11       return True
12     elif target < data[mid]:
13       # recur on the portion left of the middle
14       return binary_search(data, target, low, mid - 1)
15     else:
16       # recur on the portion right of the middle
17       return binary_search(data, target, mid + 1, high)
```

↳base case

can also return location

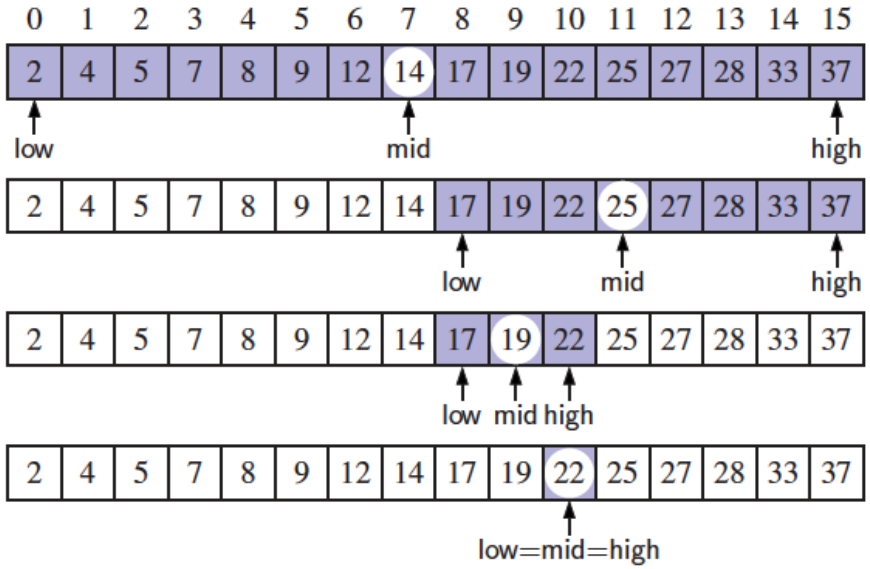
↳recursion

↳change lower bound

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:

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

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

  - 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

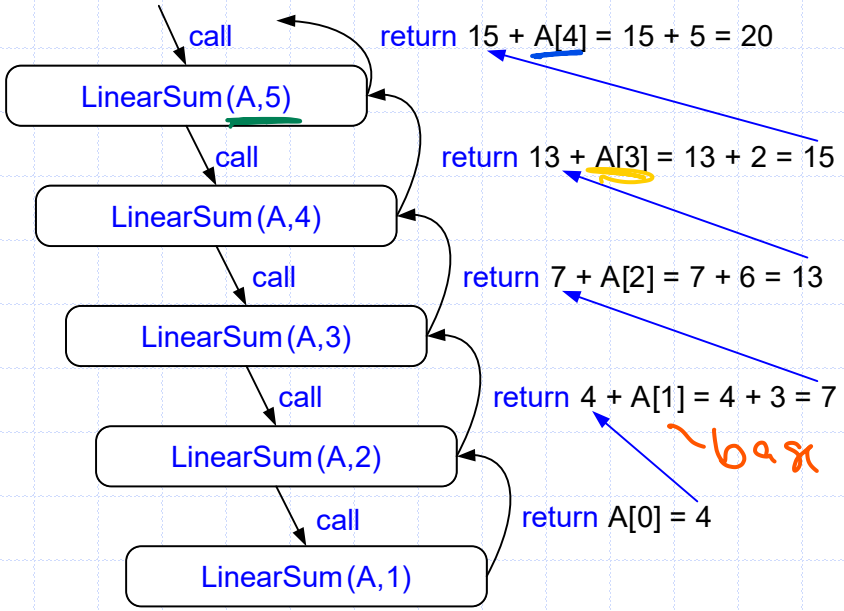
- Test for base cases
  - Begin by testing for a set of base cases (there should be at least one).
  - Every possible chain of recursive calls must eventually reach a base case, and the handling of each base case should not use recursion.
- Recur once
  - Perform a single recursive call
  - This step 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.

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:

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

Swap
↳ a,b = b,a

Computing Powers

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

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

- 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.

Recursive Squaring

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

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

- For example,

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

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

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

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

Recursive Squaring Method

Algorithm Power(x, n):

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

Output: The value x^n

if $n = 0$ **then**

return 1

if n is odd **then**

y = Power(x, (n - 1)/2)

return $x \cdot y \cdot y$

else

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

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.