To seal: moisten flap, fold over, and seal

# Recursion

Spring 2015

UW CSE 160

# Three recursive algorithms

- Sorting
- GCD (greatest common divisor)
- Exponentiation

Used in cryptography, which protects information and communication

# Sorting a list

Sir Anthony Hoare

- Python's **`sorted`** function returns a sorted version of a list.

  **`sorted([3, 1, 4, 1, 5, 9])`**

  ⇒ **`[1, 1, 3, 4, 5, 9]`**

- How could you implement **`sorted`**?
- Idea ("quicksort", invented in 1960):
  - Choose an arbitrary element (the "pivot")
  - Collect the smaller items and put them on its left
  - Collect the larger items and put them on its right

# First version of quicksort

```
def quicksort(thelist):
    """Return a sorted version of thelist."""
    pivot = thelist[0]
    smaller = [elt for elt in thelist if elt < pivot]
    larger = [elt for elt in thelist if elt > pivot]
    return smaller + [pivot] + larger

print quicksort([3, 1, 4, 1, 5, 9])
```

$\Rightarrow$ [1, 1, 3, 4, 5, 9]

There are three problems with this definition

Write a test case for each problem

# Problems with first version of quicksort

1. The “smaller” and “larger” lists aren’t sorted
2. Fails if the input list is empty
3. Duplicate elements equal to the pivot are lost

# Near-final version of quicksort

```
def quicksort(thelist):
  """Return a sorted version of thelist."""
  if len(thelist) < 2:
    return thelist
  pivot = thelist[0]
  smaller = [elt for elt in thelist if elt < pivot]
  larger = [elt for elt in thelist if elt > pivot]
  return quicksort(smaller) + [pivot] + quicksort(larger)
```

How can we fix the problem with duplicate pivot values?

# 2 ways to handle duplicate pivot values

```
def quicksort(thelist):
  """Return a sorted version of thelist."""
  if len(thelist) < 2:
    return thelist
  pivot = thelist[0]
  smaller = [elt for elt in thelist if elt < pivot]
  pivots = [elt for elt in thelist if elt == pivot]
  larger = [elt for elt in thelist if elt > pivot]
  return quicksort(smaller) + pivots + quicksort(larger)
```

```
def quicksort(thelist):
  """Return a sorted version of thelist."""
  if len(thelist) < 2:
    return thelist
  pivot = thelist[0]
  smaller = [elt for elt in thelist[1:] if elt <= pivot]
  larger = [elt for elt in thelist if elt > pivot]
  return quicksort(smaller) + [pivot] + quicksort(larger)
```

# The form of a recursive algorithm

- Determine whether the problem is small or large
- If the problem is small: ("base case")
  - Solve the whole thing
- If the problem is large: ("recursive case")
  - Divide the problem, creating one or more smaller problems
  - Ask someone else to solve the smaller problems
    - Recursive call to do most of the work
  - Do a small amount of postprocessing on the result(s) of the recursive call(s)

# Recursion design philosophy

- Recursion expresses the essence of divide and conquer
  - Solve a smaller subproblem(s), then
  - Use the answer(s) to solve the original problem

- Passing the buck:  I am willing to do a small amount of work, as long as I can offload most of the work to someone else.
- Wishful thinking:  If someone else solves most of the problem, then I will do the rest.

# Decomposition for recursion

List algorithms:

- Base case: short (or empty) list
- Recursive case: process
  - all but the first element of the list, or
    - The smaller subproblem is only a tiny bit smaller
    - The postprocessing combines the first element of the list with the recursive result
  - half of the list
    - Often recursively process both halves
    - The postprocessing combines the two recursive results

Numeric algorithms:

- Base case: small number (often 1 or 0)
- Recursive case: process a smaller value
  - 1 less than the original value
  - half of the original value
  - …

File system:

- Base case: single file
- Recursive case: process a subdirectory

Geographical algorithms:

- Base case: small area
- Recursive case: smaller part of a map (or other spatial representation)

# Recursion: base and inductive cases

- A recursive algorithm always has:
  - a base case (no recursive call)
  - an inductive or recursive case (has a recursive call)
    - solves a smaller problem
- What happens if you leave out the base case?
- What happens if you leave out the inductive case?

# Factorial

```
def fact(num):
    """ Assumes num is an int > 0, return n!"""
    if num == 1:
        return num
    else:
        return num * fact(num - 1)

print fact(3)
print fact(1)
print fact(2)
```

# Sum List

```
def sum_list(lst):
    """Returns sum of numbers in list.
    Returns zero for an empty list."""
    if len(lst) == 0:
        return 0
    else:
        return lst[0] + sum_list(lst[1:])

sum_list([1, 3, 6])
```

# Fibonacci

```
def fib(n):
    """Returns the nth Fibonacci number."""
    if n == 0 or n == 1:
        return 1
    else:
        return fib(n - 1) + fib(n - 2)

print fib(6)
```

# GCD (greatest common divisor)

gcd(a, b) = largest integer that divides both a and b

- gcd(4, 8) = 4
- gcd(15, 25) = 5
- gcd(16, 35) = 1

How can we compute GCD?

# Euclid's method for computing GCD

## (circa 300 BC, still commonly used!)

$$\gcd(a, b) = \begin{cases} a & \text{if } b = 0 \\ \gcd(b, a) & \text{if } a < b \\ \gcd(a-b, b) & \text{otherwise} \end{cases}$$

# Python code for Euclid's algorithm

```
def gcd(a, b):
  """Return the greatest common divisor of a and b."""
  if b == 0:
    return a
  elif a < b:
    return gcd(b, a)
  else:
    return gcd(a - b, b)
```

# Exponentiation

**Goal**: Perform exponentiation, using only addition, subtraction, multiplication, and division. (Example: $3^4$)

```
def exp(base, exponent):
    """Return base^exponent.
       Exponent is a non-negative integer."""
    if exponent == 0:
        return 1
    else:
        return base * exp(base, exponent - 1)
```

**Example:**

exp(3, 4)
3 * exp(3, 3)
3 * (3 * exp(3, 2))
3 * (3 * (3 * exp(3, 1)))
3 * (3 * (3 * (3 * exp(3, 0))))
3 * (3 * (3 * (3 * 1)))

# Faster exponentiation

Suppose the exponent is even.

Then, $base^{exponent} = (base*base)^{exponent/2}$

Examples: $3^4 = 9^2$ $9^2 = 81^1$ $5^{12} = 25^6$ $25^6 = 625^3$

New implementation:

```
def exp(base, exponent):
    """Return base^exponent.
       Exponent is a non-negative integer."""
    if exponent == 0:
        return 1
    elif exponent % 2 == 0:
        return exp(base * base, exponent / 2)
    else:
        return base * exp(base, exponent - 1)
```

# Comparing the two algorithms

**Original algorithm: 12 multiplications**

exp(5, 12)
5 * exp(5, 11)
5 * 5 * exp(5, 10)
5 * 5 * 5 * exp(5, 9)
…
5 * 5 * 5 * 5 * 5 * 5 * 5 * 5 * 5 * 5 * 5 * 5 * exp(5, 0)
5 * 5 * 5 * 5 * 5 * 5 * 5 * 5 * 5 * 5 * 5 * 5 * 1
5 * 5 * 5 * 5 * 5 * 5 * 5 * 5 * 5 * 5 * 5 * 5
5 * 5 * 5 * 5 * 5 * 5 * 5 * 5 * 5 * 5 * 5 * 25
5 * 5 * 5 * 5 * 5 * 5 * 5 * 5 * 5 * 5 * 125
…
244140625

**Fast algorithm: 5 multiplications**

exp(5, 12)
$(5 * 5)^6$
exp(25, 6)
$(25 * 25)^3$
exp(625, 3)
625 * exp(625, 2)
$(625 * 625)^1$
625 * exp(390625, 1)
625 * 390625 * exp(390625, 0)
625 * 390625 * 1
625 * 390625
244140625

Speed matters:
In cryptography, exponentiation is done with 600-digit numbers.

# Recursion vs. iteration

- Any recursive algorithm can be re-implemented as a loop instead
  - This is an “iterative” expression of the algorithm
- Any loop can be implemented as recursion instead

- Sometimes recursion is clearer and simpler
  - Mostly for data structures with a recursive structure
- Sometimes iteration is clearer and simpler