To seal: moisten flap, fold over, and seal

# Recursion


Michael Ernst

CSE 190p

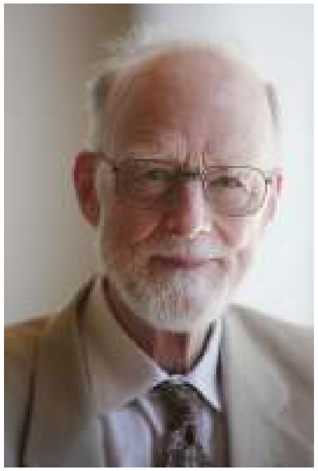
University of Washington

# 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([4, 1, 5, 2, 7])`** ⇒ **`[1, 2, 4, 5, 7]`**

- 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):
    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([4, 1, 5, 2, 7])
```

There are three problems with this definition

Write a test case for each problem

# Near-final version of quicksort

```
def quicksort(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 elements?

# Handling duplicate pivot items

```
def quicksort(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):
  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)
```

# 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):
  if b == 0:
    return a
  if a < b:
    return gcd(b, a)
  return gcd(a-b, b)
```

# Exponentiation

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

```
def exp(base, exponent):
    """Exponent is a non-negative integer"""
    if exponent == 0:
        return 1
    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, $\text{base}^{\text{exponent}} = (\text{base*base})^{\text{exponent}/2}$

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

New implementation:

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

# Comparing the two algorithms

**Original algorithm: 12 multiplications**

$5^{12}$
$5 * 5^{11}$
$5 * 5 * 5^{10}$
$5 * 5 * 5 * 5^{9}$
…
$5 * 5 * 5 * 5 * 5 * 5 * 5 * 5 * 5 * 5 * 5 * 5 * 5^{0}$
$5 * 5 * 5 * 5 * 5 * 5 * 5 * 5 * 5 * 5 * 5 * 5 * 1$
$5 * 5 * 5 * 5 * 5 * 5 * 5 * 5 * 5 * 5 * 5 * 5$
$5 * 5 * 5 * 5 * 5 * 5 * 5 * 5 * 5 * 5 * 25$
$5 * 5 * 5 * 5 * 5 * 5 * 5 * 5 * 5 * 125$
…
244140625

**Fast algorithm: 5 multiplications**

$5^{12}$
$(5 * 5)^{6}$
$25^{6}$
$(25 * 25)^{3}$
$625^{3}$

$625 * 625^{2}$
$625 * 625 * 625^{1}$
$625 * 625 * 625 * 625^{0}$
$625 * 625 * 625 * 1$
$625 * 625 * 625$
$625 * 390625$
244140625

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

# Recursion: base and inductive cases

- Recursion expresses the essence of divide and conquer
  - Solve a smaller subproblem, use the answer to solve the original problem
- A recursive algorithm always has:
  - a base case (no recursive call)
  - an inductive or recursive case (has a recursive call)
- What happens if you leave out the base case?
- What happens if you leave out the inductive case?

# 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

# More examples of recursion

- List algorithms: recursively process all but the first element of the list, or half of the list
- Map algorithms: search for an item in part of a map (or any other spatial representation)
- Numeric algorithms: Process a smaller value