COR-IS1702:
COMPUTATIONAL THINKING
WEEK 5: RECURSION

2020/21 Term 1

(05) Recursion

Video (16 mins):

https://www.youtube.com/watch?v=nHLshUOMkGw&list=PLi1cUmnkDnZvpLl1NPYxmq1Jnd7LAGCaa&index=39

Road Map

Algorithm Design and Analysis

- Week 1: Introduction, Counting, Programming
- Week 2: Programming
- Week 3: Complexity
- Week 4: Iteration & Decomposition
- This week → Week 5: Recursion

Fundamental Data Structures

(Weeks 6 - 10)

Computational Intractability and Heuristic Reasoning

(Weeks 11 - 13)

Recursion

Expressing a problem in terms of a smaller version of itself

- Recursion
  - Factorial
  - Fibonacci
- Merge Sort

Alan Perlis:
Recursion is the root of computation since it trades description for time.

Recursion

- A **recursive algorithm** is an algorithm that, as part of its operations, invokes itself over a smaller problem space.
- It is also based on the key idea of decomposition
  - breaking a large problem into smaller subproblems
- An alternative to iterative algorithms relying on loops
  - In terms of definition, recursive algorithms are simpler and more elegant
  - In terms of computation, they are not always more efficient than iteration

L. Peter Deutsch:
"*To iterate is human, to recurse divine*."

1st Example of Recursion

- Calculating **compound** interest rate
- If we place a deposit of `x` dollars with interest rate of `r` percent per year for `y` number of years, how much money do we have upon maturity?
- Solution using iteration:

x: principal

y: number of years

```
def maturity_iter(x, r, y):
  for i in range(y):
    x *= (100.0 + r)/100
  return x
```

repeat y times

…1st Example of Recursion

- Calculating **compound** interest rate
- If we place a deposit of `x` dollars with interest rate of `r` percent per year for `y` number of years, how much money do we have upon maturity?

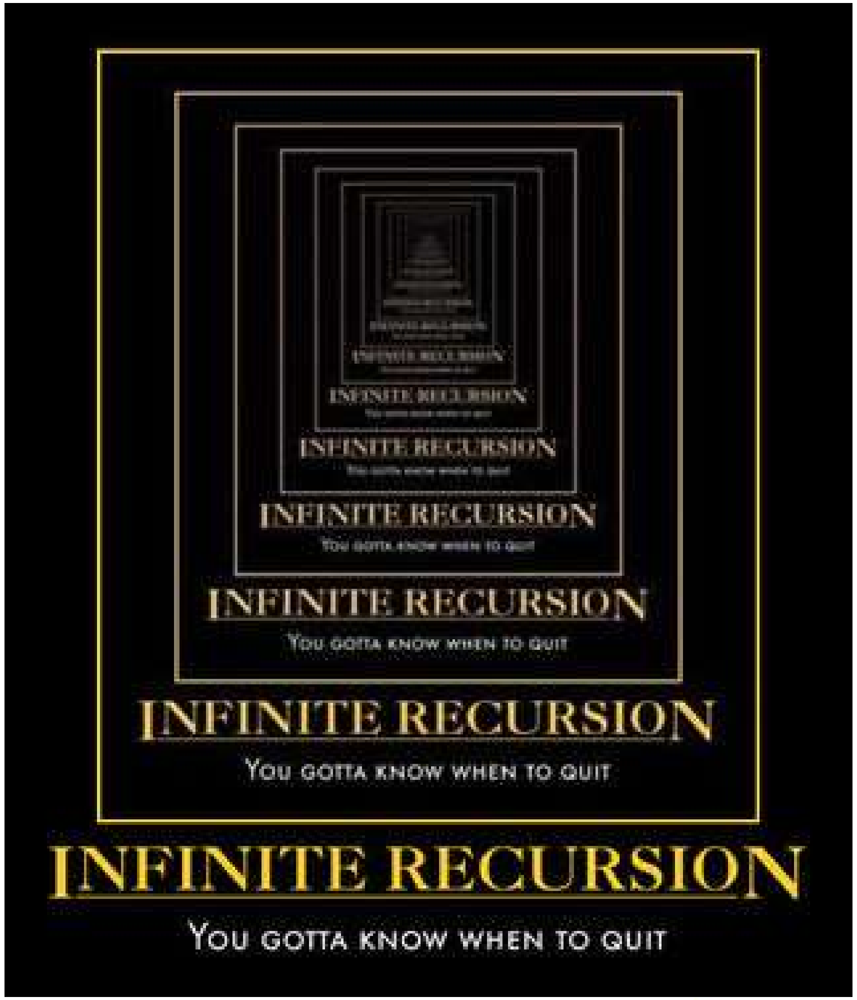
y: number of years

the function calls itself

```
def maturity(x, r, y):
  if(y == 0):
    return x
  else:
    return maturity(x, r, y-1) * (100.0 + r)/100
```

Fundamentals of Recursion

- A recursive algorithm calls itself to solve the smaller pieces
- Each recursive call should deal with a smaller instance of the same problem
  - **Reduction Step**
- There must be a stopping point, otherwise the result is infinite recursion
  - **Base Case**



http://www.peteonsoftware.com/index.php/2011/09/14/my-introduction-to-scheme-part-3/

…Fundamentals of Recursion

- **Base case**
  - the simplest possible cases that cannot be reduced anymore
- **Reduction step**
  - a set of rules that reduce other cases towards the base case

```
def maturity(x, r, y):
   if (y == 0):
      return x
   else:
      return maturity(x, r, y-1) * (100.0 + r)/100
```

base case: year 0, when interest is not yet earned

reduction step: this year’s amount is last year’s plus interest

2nd Example of Recursion: Factorial

- Compute the factorial `n`! of an integer `n`
- `n!` = `n` x `(n-1)` x `(n-2)` x ... x `1`
  - e.g. 4! = 4 x 3 x 2 x 1
- **Reduction step:** `n!` = `n` x `(n-1)!`
  - $4! = 4 \times \boxed{3!}$

    $= 4 \times \boxed{3 \times \boxed{2!}}$

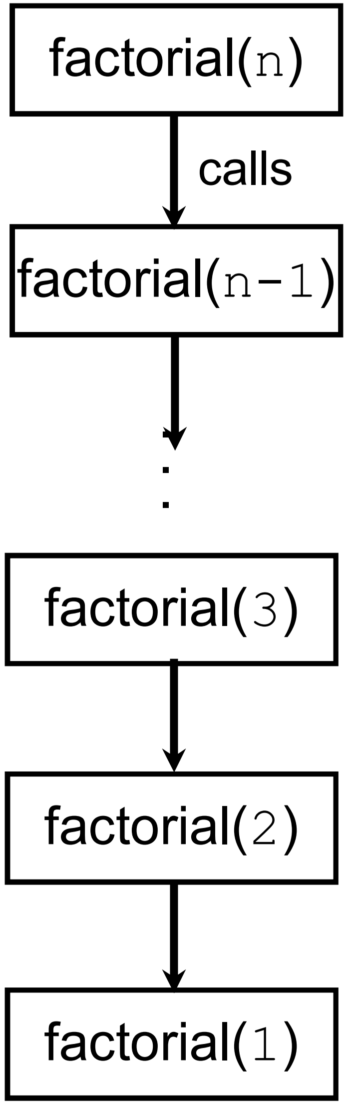
    $= 4 \times \boxed{3 \times \boxed{2 \times 1!}}$
- **Base case:** 1! = 1

Recursive Algorithm for Factorial

- Let's write a recursive algorithm `factorial(n)` to compute `n`!
- **Reduction step:**
  - `factorial(n)` = `n` x `factorial(n-1)`
- **Base case:**
  - `factorial(1)` = `1`

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

Tracing Recursive Calls



See example:
http://cs.nyu.edu/courses/spring07/V22.0101-002/19slide.ppt
(slides 12-22)

Iterative vs. Recursive for Factorial

Iterative

```
def factorial(n):
  f = 1
  i = n
  while i > 0:
    f = f * i
    i = i - 1
  return f
```

Recursive

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

Worst-case complexity is O(n) for both iterative and recursive algorithms.

3rd Example of Recursion: Fibonacci

- Fibonacci series: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ...
- Each number is the sum of the previous two numbers
  - `fibonacci(0) = 0`
  - `fibonacci(1) = 1`

    fixed (by definition)
  - `fibonacci(2) = 1 + 0 = 1`
  - `fibonacci(3) = 1 + 1 = 2`
  - `fibonacci(4) = 2 + 1 = 3`
  - `fibonacci(5) = 3 + 2 = 5`
  - `...`
- **Reduction**: `fibonacci(n) = fibonacci(n-1)+fibonacci(n-2)`
- **Base cases**: `fibonacci(0) = 0, fibonacci(1) = 1`

there can be more than one base case

Recursive Algorithm for Fibonacci

- **Reduction**:
  - `fibonacci(n) = fibonacci(n-1)+fibonacci(n-2)`
- **Base cases**:
  - `fibonacci(0) = 0, fibonacci(1) = 1`

```
def fibonacci(n):
  if n == 0:
    return 0
  elif n == 1:
    return 1
  else:
    return fibonacci(n-1) + fibonacci(n-2)
```

base cases

every invocation leads to two recursive calls
with similar problem size ~ n

Tracing Recursive Calls

`fibonacci(0)`

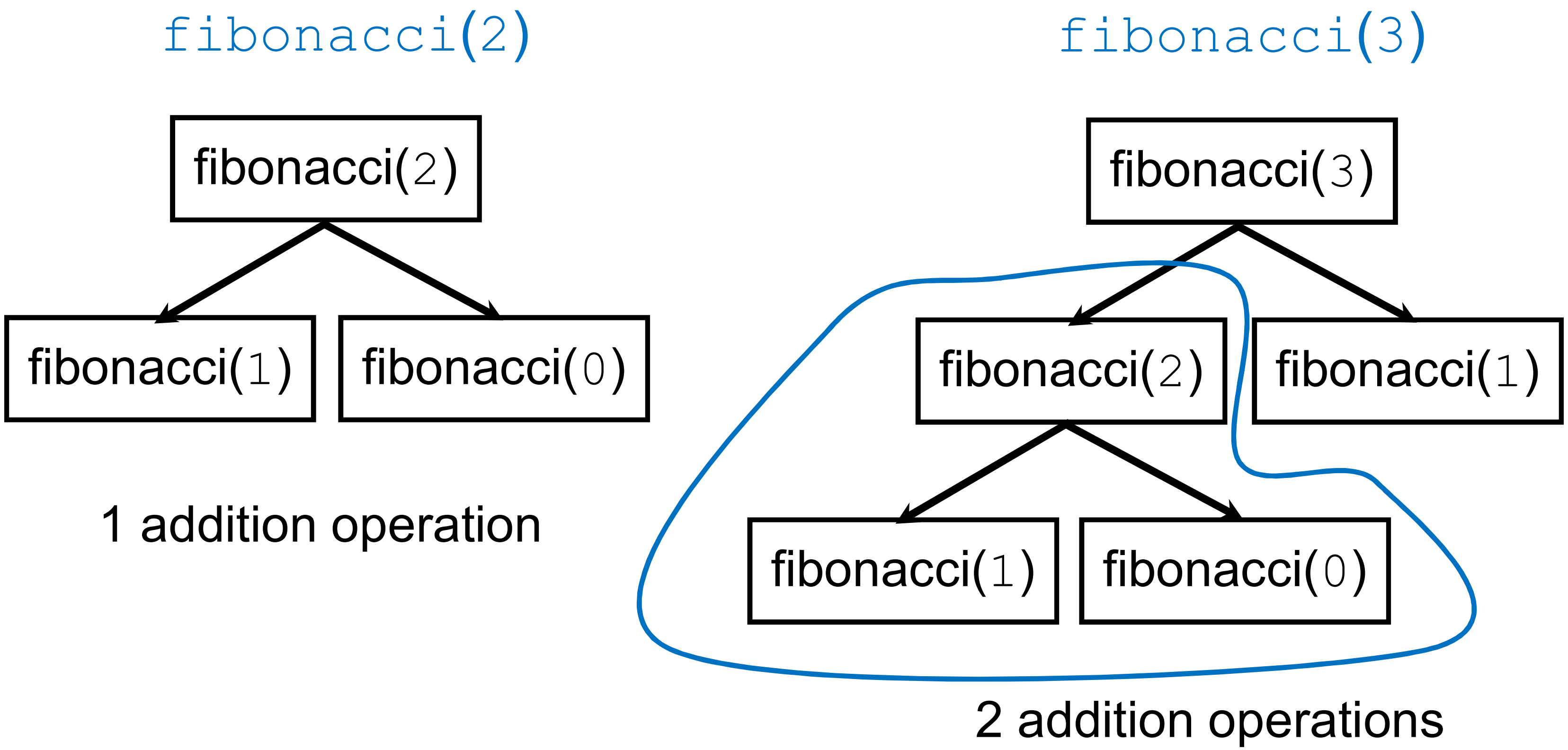
fibonacci(0)

`fibonacci(1)`

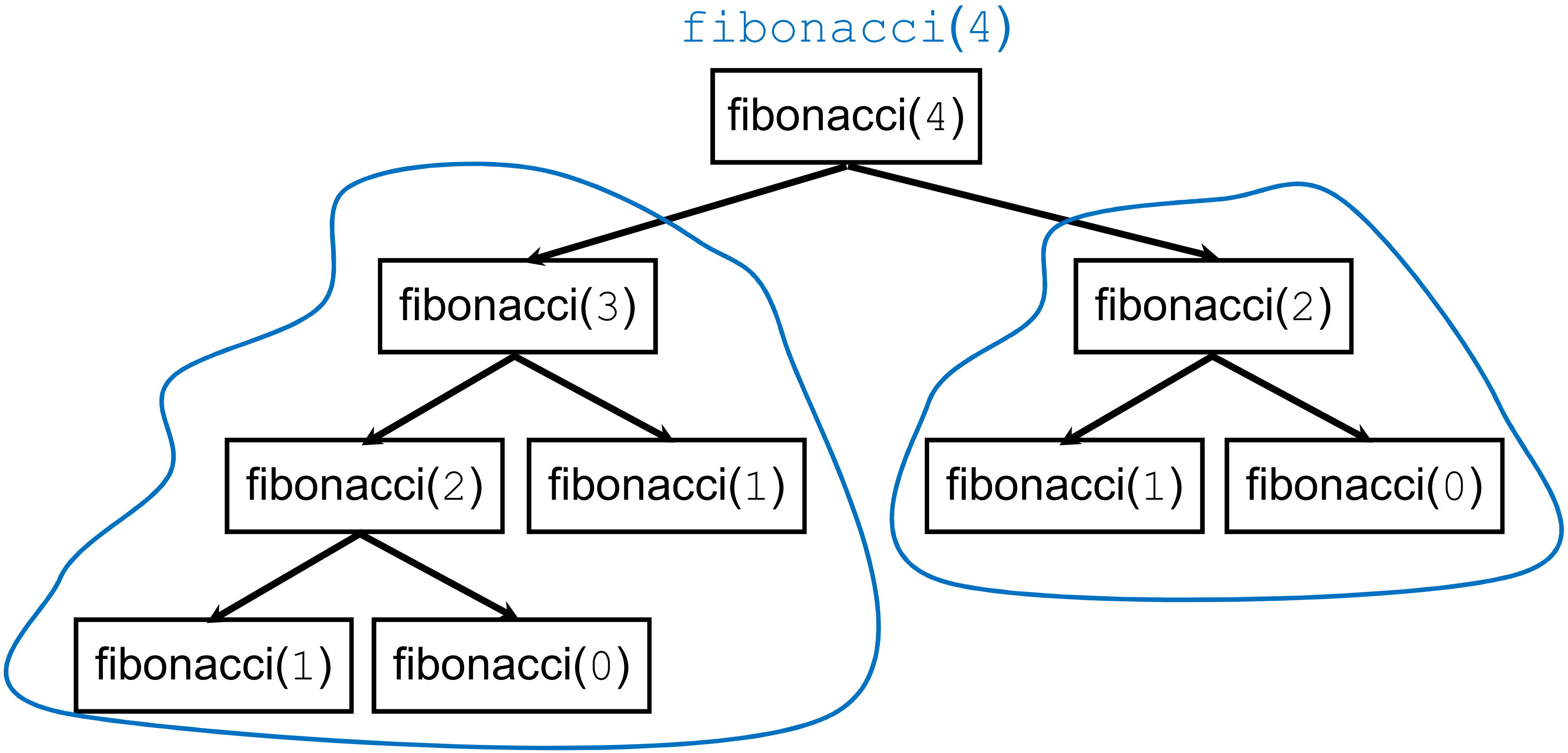
fibonacci(1)

Base cases do not lead to further recursive calls

Tracing Recursive Calls



Tracing Recursive Calls



no. of additions for `fibonacci(3)` + no of additions for `fibonacci(2)` + 1

In general, no. of additions for `fibonacci(n)` =
no. of additions for `fibonacci(n-1)` + no of additions for `fibonacci(n-2)` + 1

No. of addition operations for increasing `n`

n	No of addition operations	Ratio of additions for n to additions for n-1
0	0	n.a.
1	0	n.a.
2	1	n.a.
3	2	2
4	4	2
5	7	1.75
6	12	1.71
7	20	1.67
8	33	1.65
9	54	1.64
10	88	1.63
50	2×10^{10}	1.62
100	6×10^{20}	1.62

Complexity of Recursive `fibonacci(n)`

- In the limit, as `n` increases by 1, the number of addition operations almost double (approximately `1.6` times).
- The number of operations is approximately 1.6^n.
- Big O is an “upper bound” concept, and $1.6^n < 2^n$
- Worst-case complexity is $O(2^n)$, i.e., exponential.
  - $O(1.6^n)$ is also correct, and is in fact a tighter bound
  - For simplicity, we use $O(2^n)$.

Recursion Not Necessarily More Efficient

Iterative version of `fibonacci(n)`

```
def fibonacci(n):
  if n == 0:
    return 0
  x = 0                          #base case fibonacci(0)
  y = 1                          #base case fibonacci(1)
  for i in range(2, n+1):        #fibonacci(2) and so on
    z = x + y                    #sum the previous two numbers
    x = y                  #shift x, y to most recent 2 numbers
    y = z
  return y
```

The iterative algorithm has a single `for` loop, running `n-1` times. Worst-case complexity is O(`n`), much less than the recursive version.

In-Class Exercise Q1

- Problem: to find $\mathbf{x^n}$ (for n > 0)
- 1st way to do this:

$$x^0 = 1$$
$$x^n = x * x^{n-1}$$

e.g.

$$\begin{aligned} x^4 &= x * \boxed{x^3} \\ &= x * \boxed{x * \boxed{x^2}} \\ &= x * x * \boxed{x * \boxed{x^1}} \\ &= x * x * x * \boxed{x * x^0} \\ &= x * x * x * x * 1 \end{aligned}$$

…In-Class Exercise Q1

$$x^n = x * x^{(n-1)}$$

```
def power1(x, n):
  if n == 0:
    return 1
  else:
    return x * power1(x, n-1)
```

a. How many multiplications will the function **power1** perform when **x** = 3 and **n** = 16?
b. What is the complexity of **power1**?

In-Class Exercise Q2

- 2nd way to compute $\mathbf{x^n}$:

$$x^0 = 1$$

$$\textit{case 1 (n is even)}: \quad x^n = x^{n//2} * x^{n//2}$$

$$\textit{case 2 (n is odd)}: \quad x^n = x * x^{n//2} * x^{n//2}$$

9 and 1 are odd. Use case 2 formula.

e.g.

$$3^9 = 3 * \boxed{3^4} * \boxed{3^4}$$

4, 2 are even. Use case 1 formula.

$$= 3 * \boxed{3^2 * 3^2} * \boxed{3^2 * 3^2}$$

$$= 3 * \boxed{\boxed{3^1} * 3^1 * 3^1 * 3^1} * \boxed{3^1 * 3^1 * 3^1 * 3^1}$$

$$\boxed{\begin{aligned} 3^1 &= 3 * 3^0 * 3^0 \\ &= 3 * 1 * 1 \\ &= 3 \end{aligned}}$$

…In-Class Exercise Q2

```
def power2(x, n):
  if n == 0:
    return 1
  elif n % 2 == 0:   # n is even
    return power2(x, n//2) * power2(x, n//2)
  else:              # n is odd
    return x * power2(x, n//2) * power2(x, n//2)
```

a. How many <u>multiplications</u> will the function **power2** perform when **x** = 3 and **n** = 16?
b. What is the complexity of **power2**?

(05) Solutions to in-class Ex Q1 & Q2

Video (12 mins):

https://www.youtube.com/watch?v=HS8RDpwNog8&list=PLi1cUmnkDnZvpLI1NPYxmq1Jnd7LAGCaa&index=40