Intro to Computer Science

Previous

- Functions (continued)

Readings	
Gaddis	• Chapter 5

Next

- Recursion

Readings	
Gaddis	• Chapter 12

Recursion

- **Recursion is more than a programming topic**
  - "One of the central ideas of computer science"

- Simply: a function that calls itself
- More accurately: a method of problem solving

Recursion (is simple!)

- Essentially two aspects to a recursive function
  1. A base case
  2. A set of rules that reduce all other cases toward the base case
- The "difficult" part is learning to *think* this way

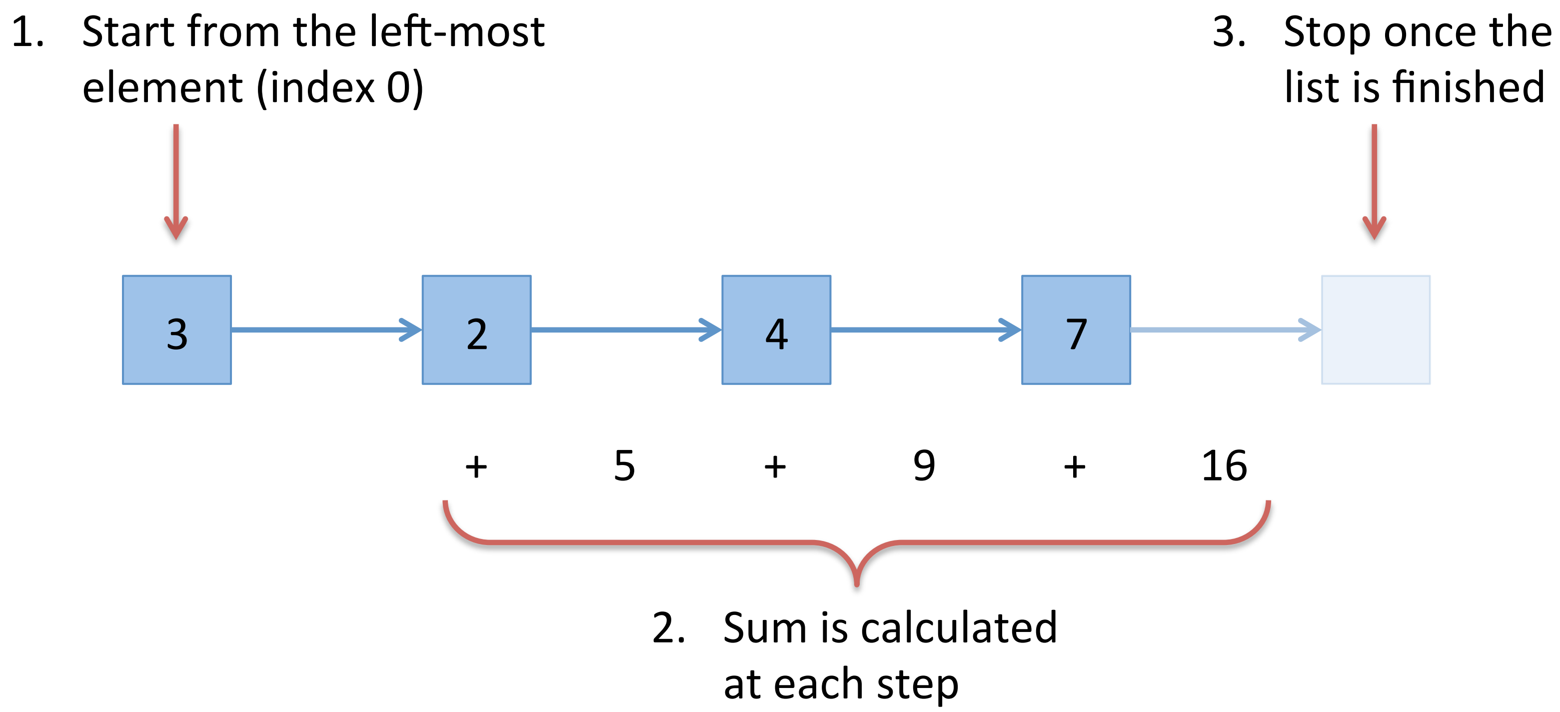
Example: list addition

The problem

- Your given a list of numbers
- Find the sum

Iterative addition

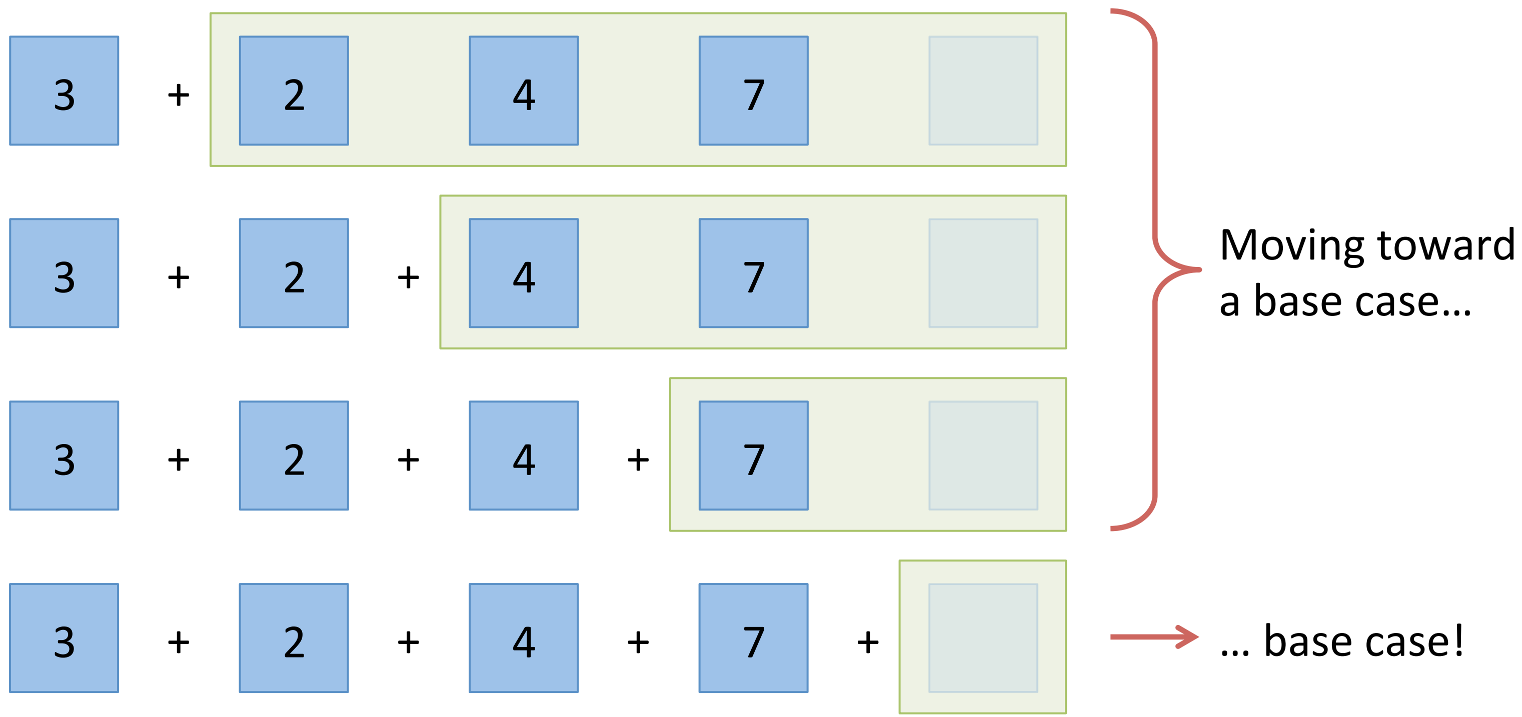
- The iterative thinker sequentially accumulates the sum



Recursive addition

The recursive thinker realizes list summation is

- The head of the list added-to *the rest of the list*
- The empty list is the identity element (zero)



Example: factorial

- Factorial: the product of all positive integers less-than or equal-to a given integer
- We assume the integer is positive

Factorial: the iterative approach

- The iterative thinker models the product over a descending *list* of integers

$$5! = 5 \times 4 \times 3 \times 2 \times 1$$

$$n! = \prod_{k=1}^{n} k$$

Factorial: the recursive approach

The recursive thinker realizes

- The integer, multiplied by the factorial of one-less
- Zero-factorial is one

$$
\begin{aligned}
5! &= 5\times 4! \\
&= 5\times 4\times 3! \\
&= 5\times 4\times 3\times 2! \\
&= 5\times 4\times 3\times 2\times 1! \\
&= 5\times 4\times 3\times 2\times 1\times 0!
\end{aligned}
$$

Getting closer a base case…

… base case!

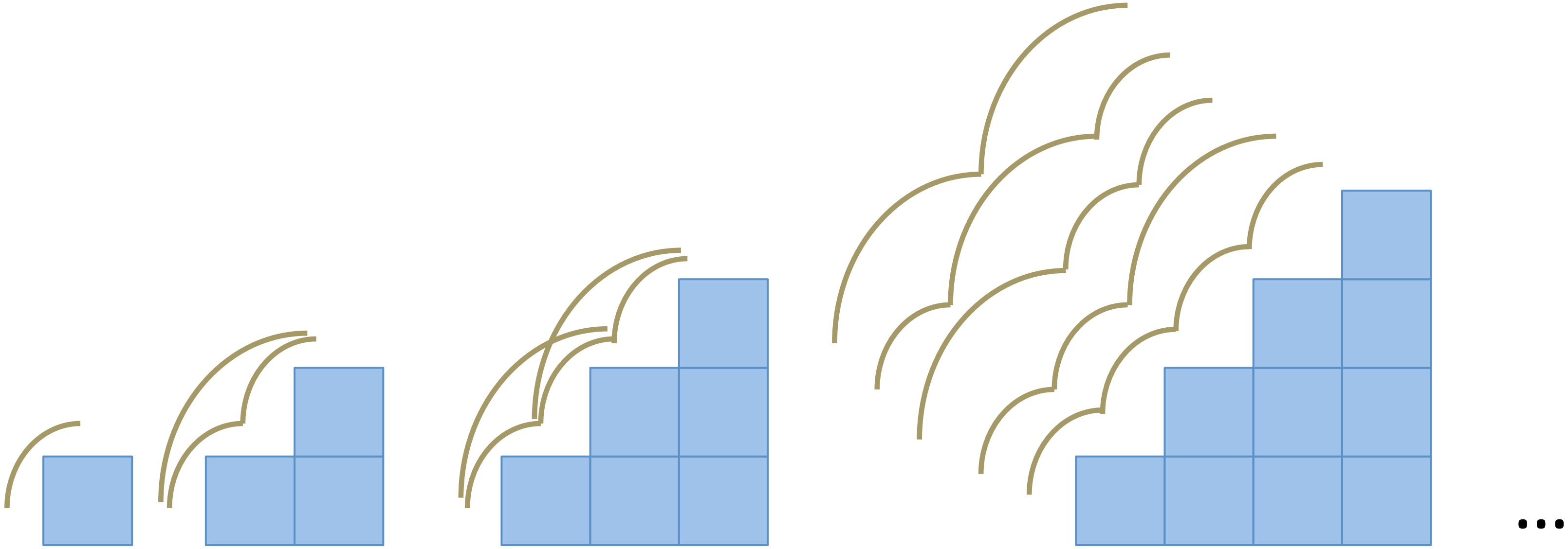
Example: party stairs

The problem:

- John has just left a raging party
- To reach his room he has 10 stairs to climb
- Because he is still in "party mode," he will take one or two stairs at a time
- How many possible paths are there to his room?

No party like an iterative party

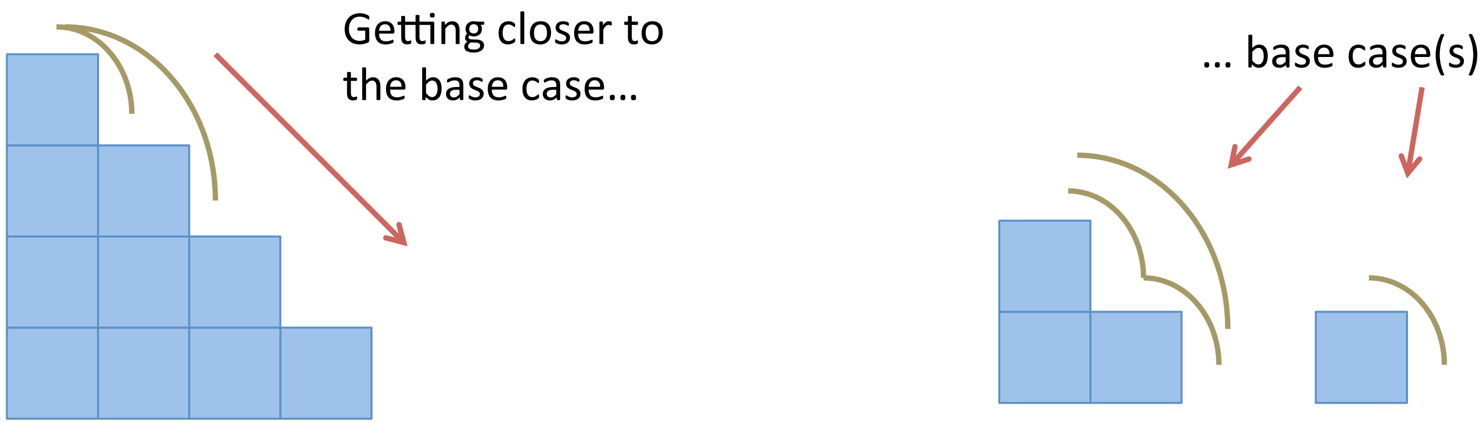
- The iterative thinker enumerates the possibilities (and then probably starts to go crazy)



stairs:	1	2	3	4
routes:	1	2	3	5

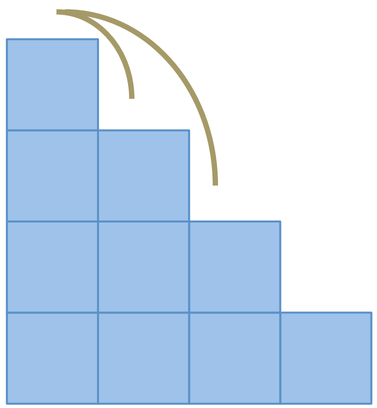
No party like a recursive party

- The recursive thinker realizes
  - There are two ways to reach the final step
  - There are two possibilities to start the climb



No party like a recursive party

- The recursive thinker realizes
  - There are two ways to reach the final step
  - There are two possibilities at the start



Reaching the final step:

1. From b_9
2. From b_8

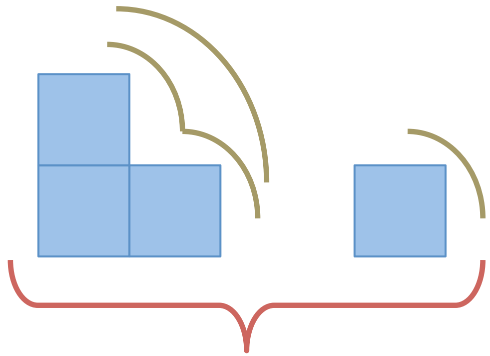
Thus, $b_{10} = b_8 + b_9$

b_6+b_7 $\quad$ b_8+b_7

...

b_4+b_5 $\quad$ b_5+b_6

... $\quad$ ...



Two ways to begin:

1. $b_1 = 1$
2. $b_2 = 2$

Blocks can contain anything

- Recall our syntactic definitions

```
for i in sequence:
    block
```

```
if expression:
    block
```

```
def function(args):
    block
```

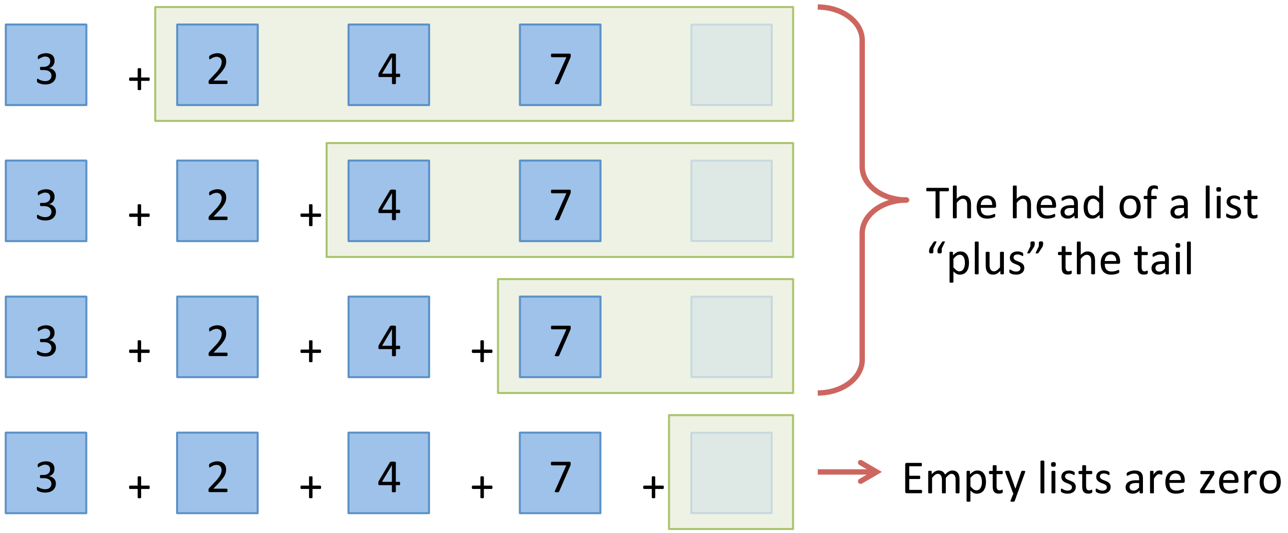
- Within blocks we sometimes added calculations
- But sometimes we added more constructs
- *We can do the same with functions*

```
for i in range(10):
    for j in range(10):
        print(i, j)
```

```
if x > y:
    if a > b:
        print(x, b)
```

```
def myFun(x):
    myFun(x)
```

Recursive addition




```
def rsum(h):
    if not h:
        return 0
    else:
        return h[0] + rsum(h[1:])
```

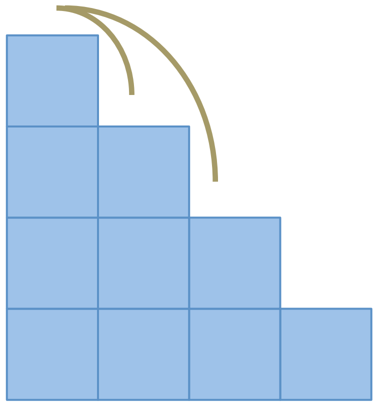
Factorial: the recursive approach

$$\left.\begin{aligned}5! &= 5\times 4!\\ &= 5\times 4\times 3!\\ &= 5\times 4\times 3\times 2!\\ &= 5\times 4\times 3\times 2\times 1!\end{aligned}\right\}\text{Getting closer a base case...}$$

$$= 5\times 4\times 3\times 2\times 1\times 0! \rightarrow \text{... base case!}$$

```
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n-1)
```

No party like a recursive party



Reaching the final step:

1. From b_9
2. From b_8

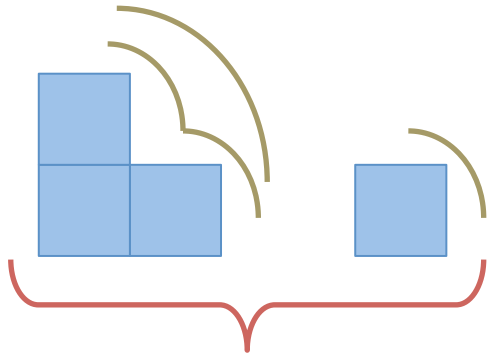
Thus, $b_{10} = b_8 + b_9$

b_6+b_7 $\quad$ b_8+b_7

…

b_4+b_5 $\quad$ b_5+b_6

… $\quad$ …



Two ways to begin:

1. $b_1 = 1$
2. $b_2 = 2$

```
def stairs(n):
    if n > 2:
        return stairs(n-1) + stairs(n-2)
    else:
        return n
```

Practicalities of recursion

- Recursion is powerful!
- But it can get nasty
  - "Loops gone wild"

➢ To fully understand how recursion works in practice, you must understand one more detail about functions

Calling a function: what really happens

- When we call a function
  - Values are assigned to local variables
  - Python makes a note of where to return
  - Execution moves to the function
- We glossed over the details!
  - Recursion requires an understanding of the details

```
def second(x, y):
    print(x, y)
    return

def first(x, y):
    second(x, y)
    return

first(0, 1)
```

Calling a function: what really happens

- Parameters, local variables, and return locations all require memory
- This allocated memory is called a *stack frame*
  - Parameters
  - Local variables
  - Return location

Calling a function: what really happens

```
def second(x, y):
    print(x, y)
    return

def first(x, y):
    second(x, y)
    return

first(0, 1)
```

Memory is allocated as we call more functions

Memory is de-allocated when the function is returned

(main)

first:
- params: 0, 1
- locals: none
- return: main

(main)

second:
- Params: 0, 1
- locals: none
- return: first

first:
- params: 0, 1
- locals: none
- return: main

(main)

first:
- params: 0, 1
- locals: none
- return: main

(main)

(main)

Calling a function: what really happens

- In typical usage, this doesn't matter much
  - It's very difficult to, by hand, call enough functions to stress the memory
- But with recursion…

Make it stop!

```
def recurse(x):
    recurse(x + 1)

recurse(0)
```

...

recurse:
- params: 2
- locals: none
- return: main

recurse:
- params: 1
- locals: none
- return: main

recurse:
- params: 0
- locals: none
- return: main

(main)

Computer fires are not good

1. Establish a base case
2. If you don’t hit the base case, ensure that you’re recursion is making progress toward that base case

Recursion versus iteration

- Many problems that can be solved recursively can also be solved iteratively (with a loop)
- Iteration is often preferred in practice
  - Reasoning about it is “more straightforward”
  - Practical aspects of memory consumption
- Python has a relatively small limit on recursive calls (~1000 frames)
  - The language *favors* iteration

However!

- Recursion has a (subjective) elegance ☺
  - Separates the hacker from the artist
- Iteration relies on assignment
  - Some languages don't have assignment!
- Many problems lend themselves better to recursive solutions (next class)
- *Regardless of your implementation, being able to think recursively is a powerful tool*