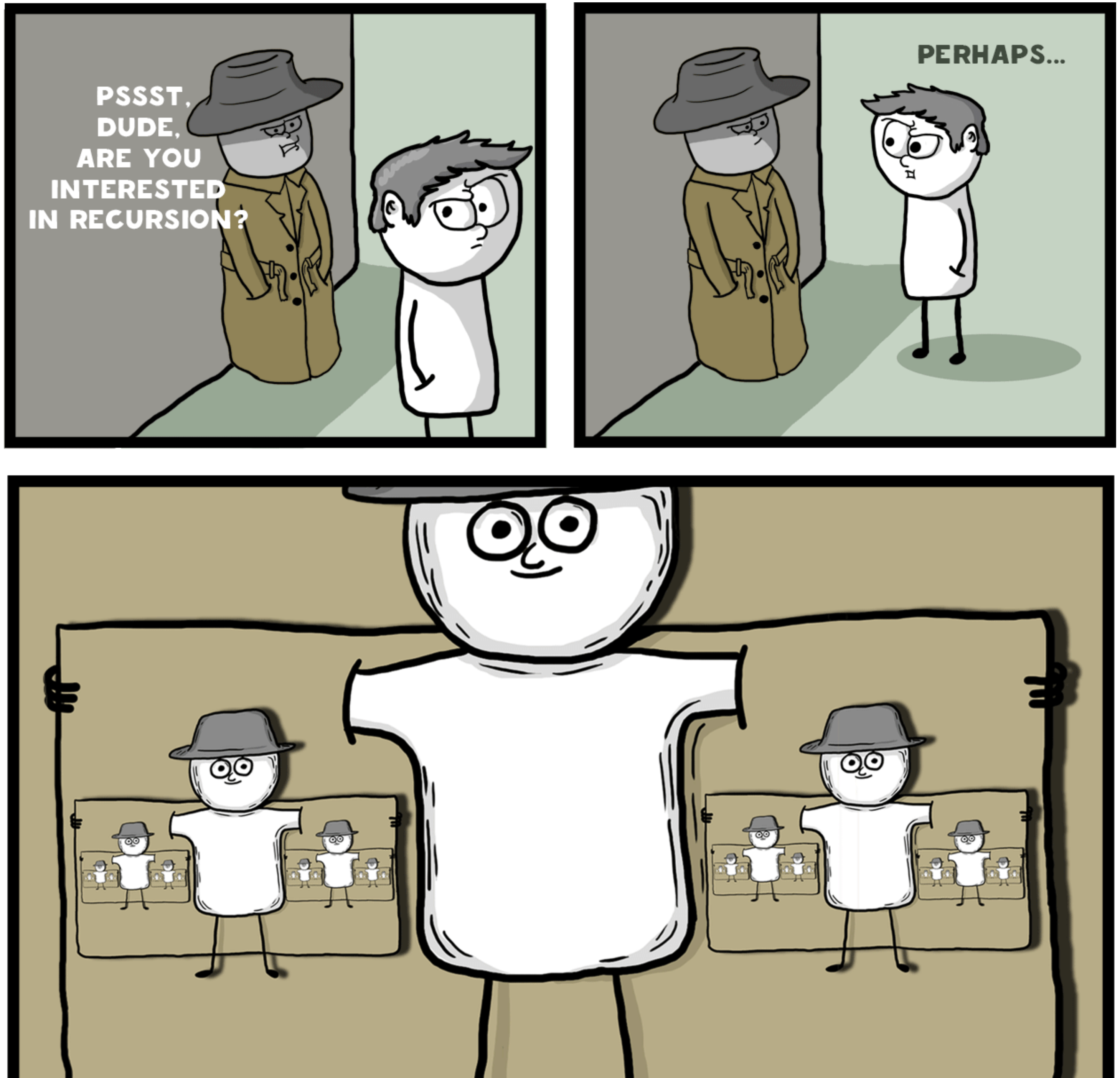
Lesson 17

Recursion

Lesson Objectives

- understand the concept of recursion
- what is a base case
- avoid infinite recursion
- recursion with more than one base case
- recursion with non - numerics


PSSST, DUDE, ARE YOU INTERESTED IN RECURSION?
PERHAPS...

What is recursion ?

- Algorithmically: a way to design solutions to problems by **divide-and-conquer** or **decrease-and-conquer**
  - reduce a problem to simpler versions of the same problem
- Semantically: a programming technique where a **function calls itself**
  - in programming, goal is to NOT have infinite recursion
    - must have **1 or more base cases** that are easy to solve
    - must solve the same problem on **some other input** with the goal of simplifying the larger problem input

Iteration so far . . .

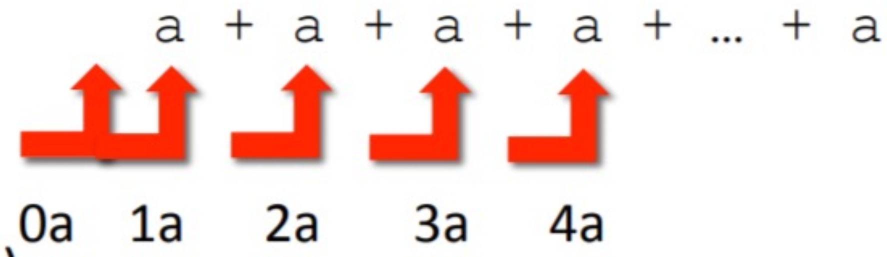
- looping constructs (`while` and `for` loops) lead to **iterative** algorithms
- can capture computation in a set of **state variables** that update on each iteration through loop

Multiplying using iteration

- "multiply a * b" is equivalent to "add a to itself b times"
- capture **state** by
  - an **iteration** number (i) starts at b

    i ← i-1 and stop when 0
  - a current **value of computation** (result)

    result ← result + a




```
def mult_iter(a, b):
    result = 0
    while b > 0:
        result += a
        b -= 1
    return result
```

iteration

current value of computation, a running sum

current value of iteration variable

Multiplying using recursion

- **recursive step**
  - think how to reduce problem to a **simpler/ smaller version** of same problem
- **base case**
  - keep reducing problem until reach a simple case that can be **solved directly**
  - when b = 1, a*b = a

```
a*b = a + a + a + a + … + a      (b times)

    = a + a + a + a + … + a      (b-1 times)

    = a + a * (b-1)              (recursive reduction)
```

```
def mult(a, b):
    if b == 1:
        return a
    else:
        return a + mult(a, b-1)
```

base case

recursive step

Example : Factorial – Demo First

`n! = n*(n-1)*(n-2)*(n-3)* … * 1`

- for what `n` do we know the factorial?

  n = 1 →

```
if n == 1:
    return 1
```

  base case

- how to reduce problem? Rewrite in terms of something simpler to reach base case

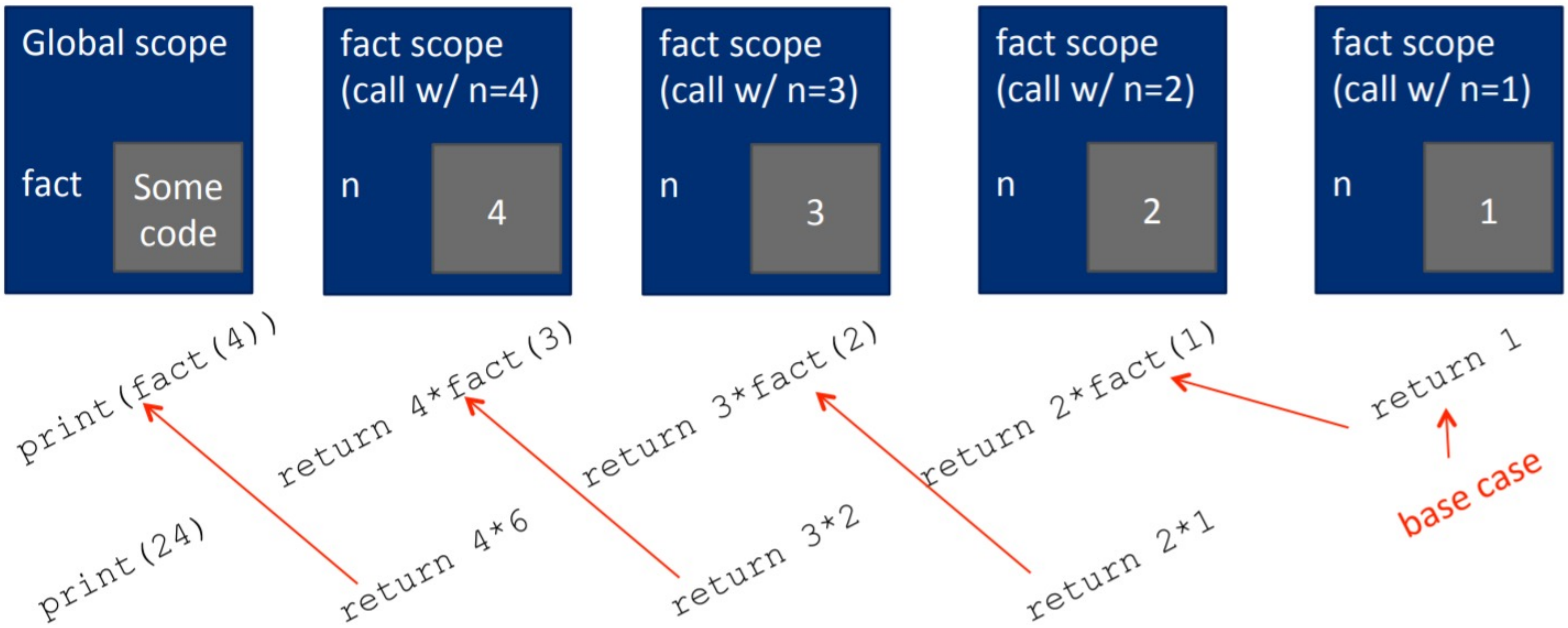
  n*(n-1)! →

```
else:
    return n*factorial(n-1)
```

  recursive step

Factorial – Tracing it out

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

print(fact(4))
```



Important to note . . .

- each recursive call to a function creates its **own scope/environment**
- **bindings of variables** in a scope are not changed by recursive call
- flow of control passes back to **previous scope** once function call returns value

using the same variable names but they are different objects in separate scopes

Iteration VS Recursion

```
def factorial_iter(n):
    prod = 1
    for i in range(1,n+1):
        prod *= i
    return prod
```

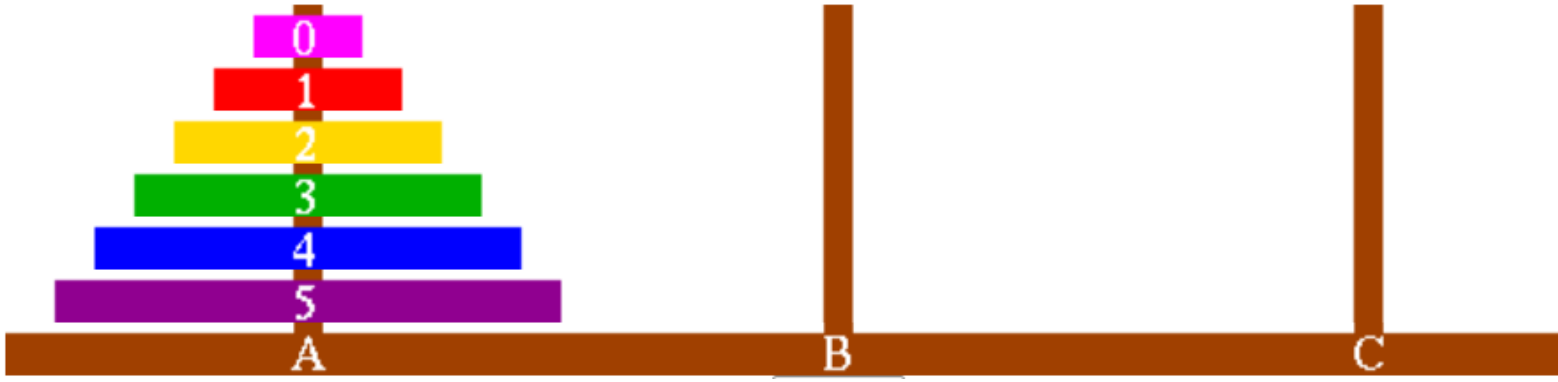
```
def factorial(n):
    if n == 1:
        return 1
    else:
        return n*factorial(n-1)
```

We will talk more about efficiency in the topic of searching and sorting.

- recursion may be simpler, more intuitive
- recursion may be efficient from programmer POV
- recursion may not be efficient from computer POV

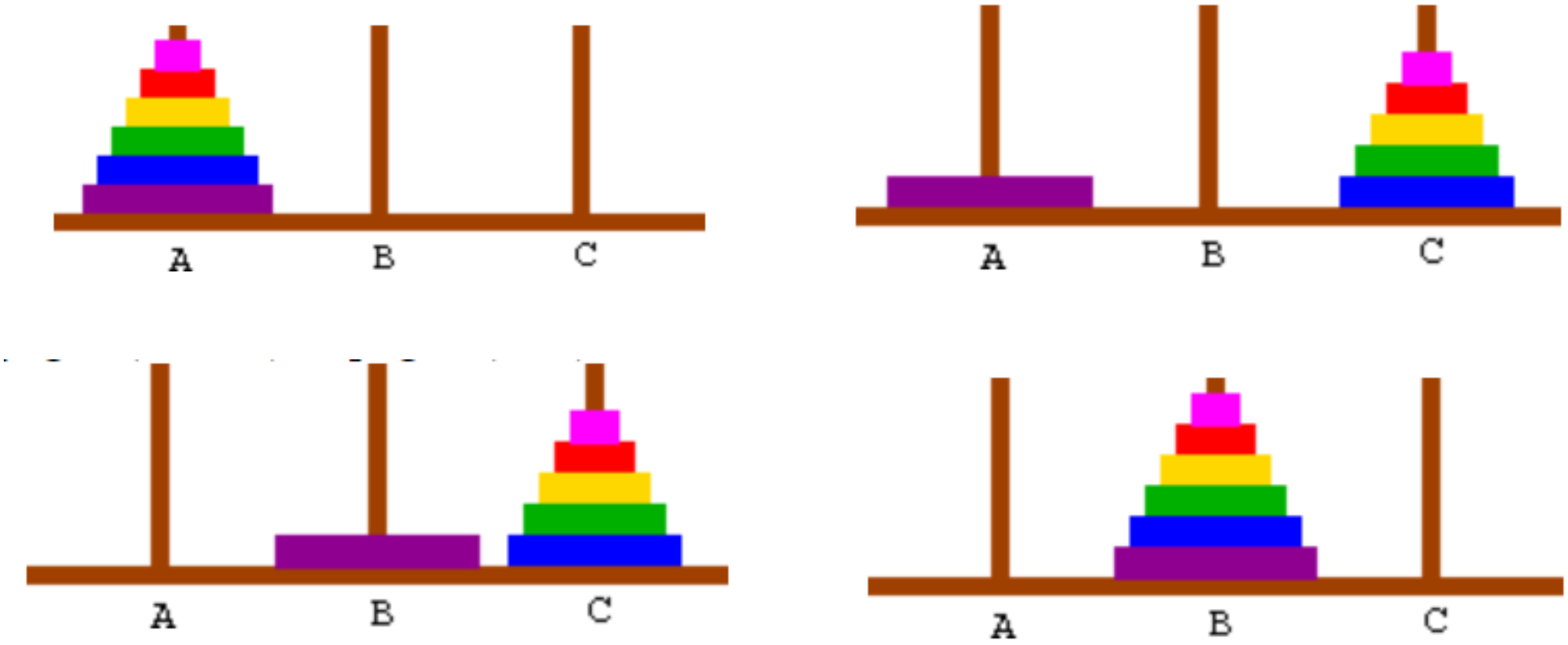
Tower of Hanoi

1. Our goal is to move the entire tower to the middle peg.
2. We can only move one disk at a time.
3. We can never place a larger disk on a smaller one.



- Difficult to solve through Iteration
- Easy when Recursion is used
- Try it yourself : https://www.mathsisfun.com/games/towerofhanoi.html

Think Recursively



Tower of Hanoi : the code

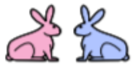
Invest time to really digest and understand this code !

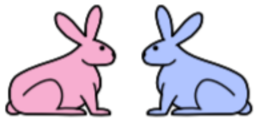
```
def printMove(fr, to):
    print('move from ' + str(fr) + ' to ' + str(to))


def Towers(n, fr, to, spare):
    if n == 1:
        printMove(fr, to)
    else:
        Towers(n-1, fr, spare, to)
        Towers(1, fr, to, spare)
        Towers(n-1, spare, to, fr)
```

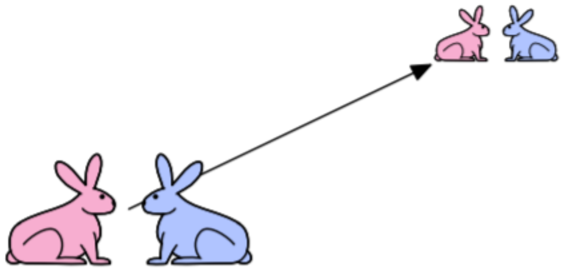
```
>>> Towers(4,1,2,3)
move 1 to 3
move 1 to 2
move 3 to 2
move 1 to 3
move 2 to 1
move 2 to 3
move 1 to 3
move 1 to 2
move 3 to 2
move 3 to 1
move 2 to 1
move 3 to 2
move 1 to 3
move 1 to 2
move 3 to 2
```

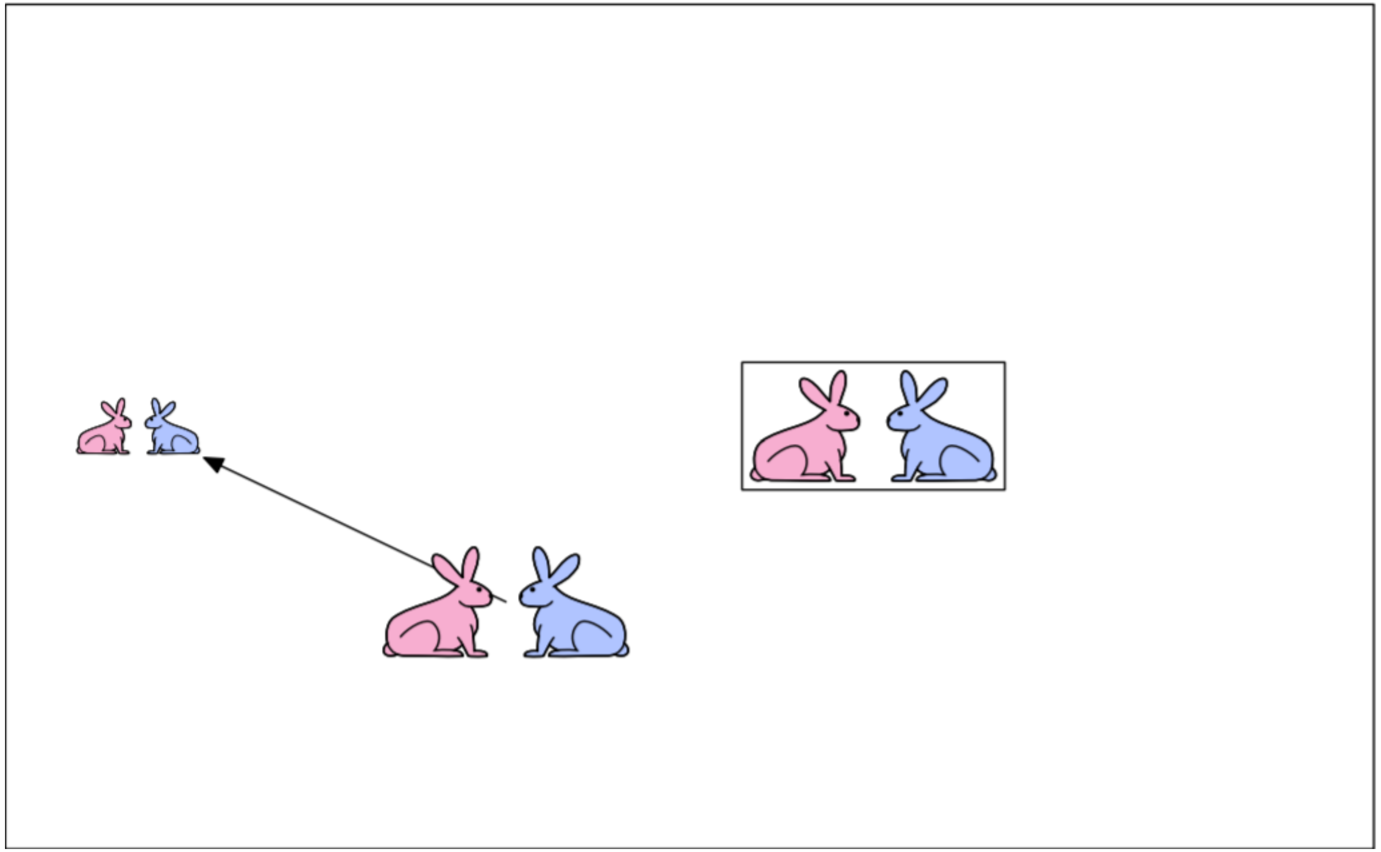
Example : Fibonacci Number

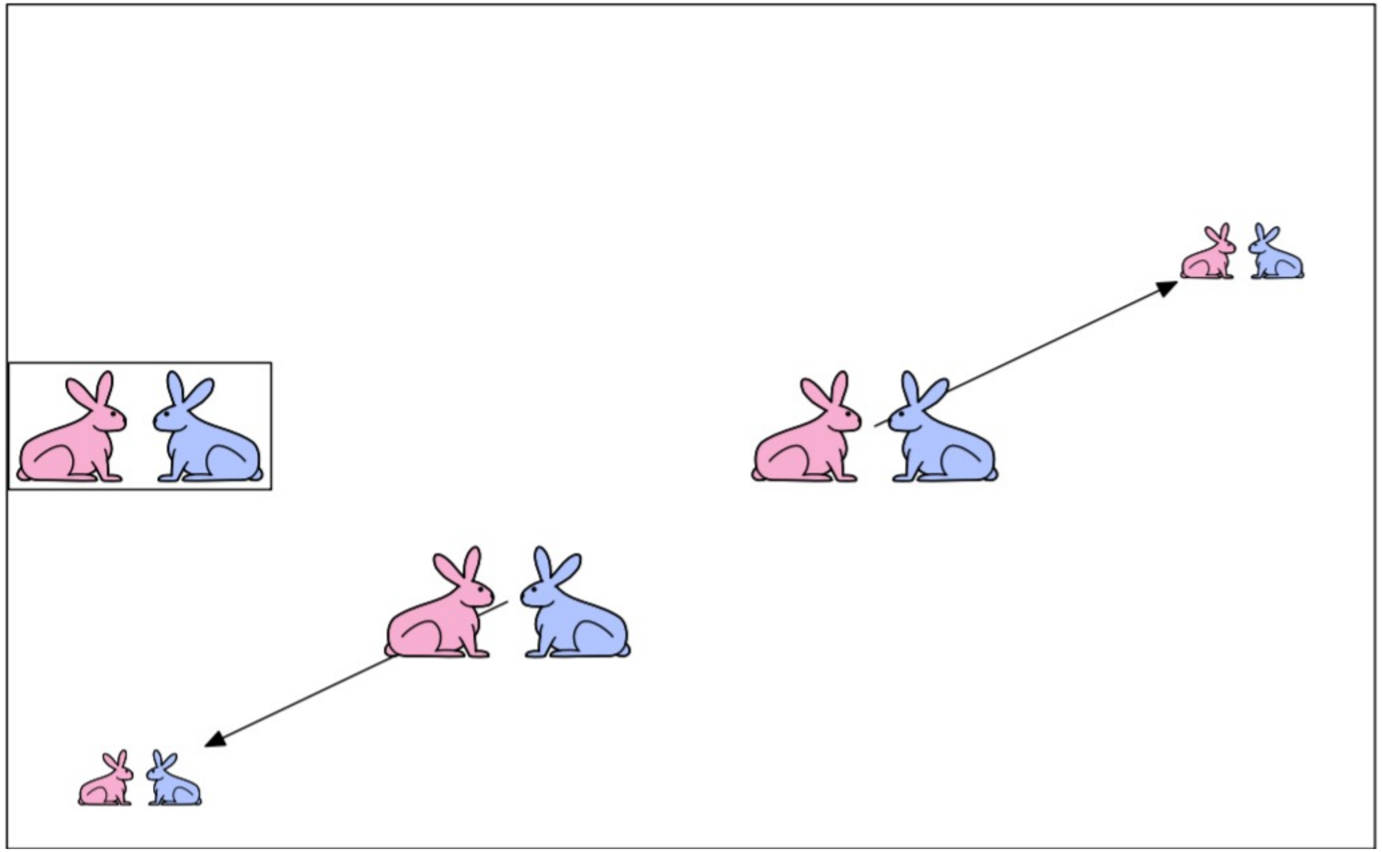
- Fibonacci numbers
  - Leonardo of Pisa (aka Fibonacci) modeled the following challenge
    - Newborn pair of rabbits (one female, one male) are put in a pen
    - Rabbits mate at age of one month
    - Rabbits have a one month gestation period
    - Assume rabbits never die, that female always produces one new pair (one male, one female) every month from its second month on.
    - How many female rabbits are there at the end of one year?

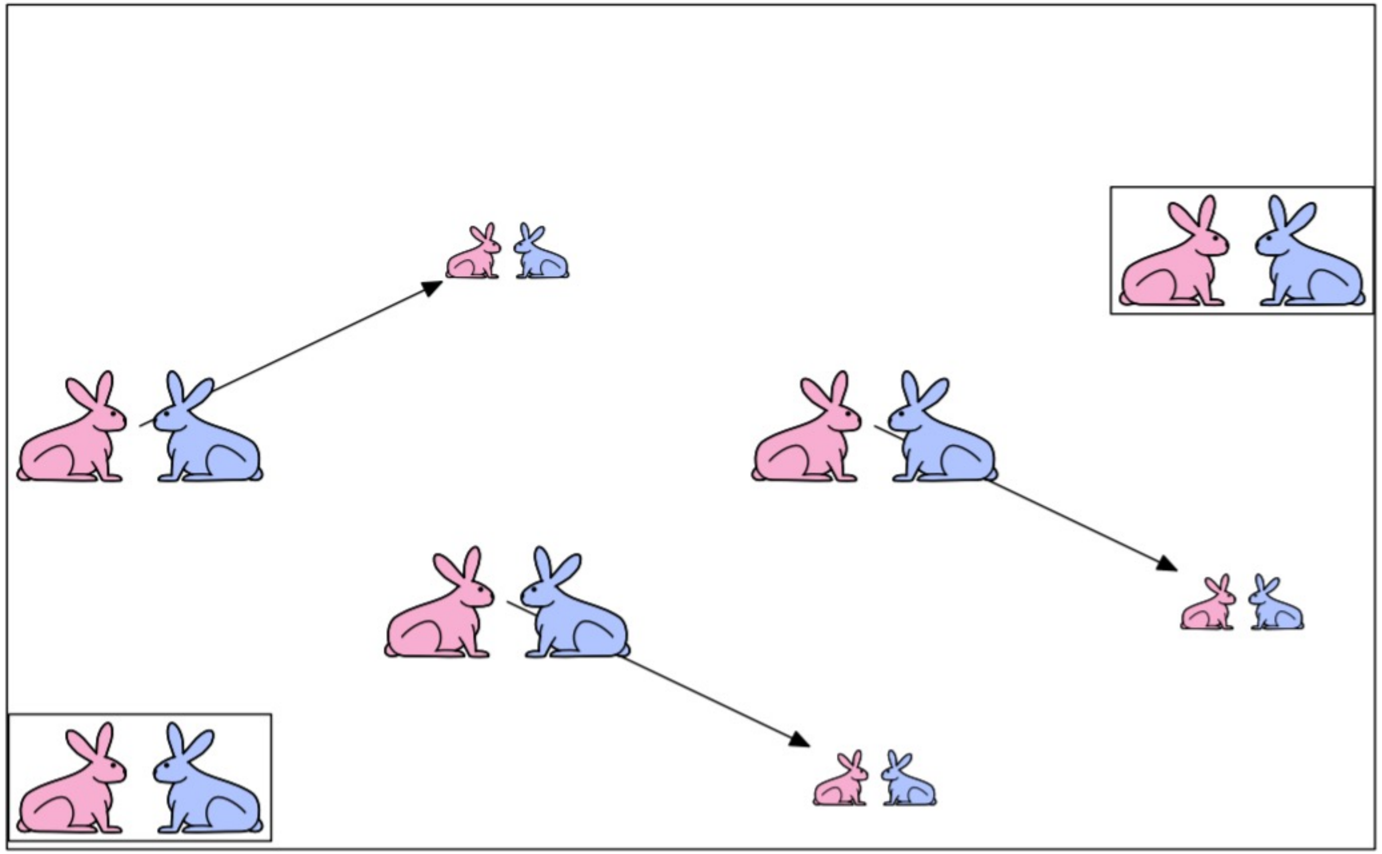


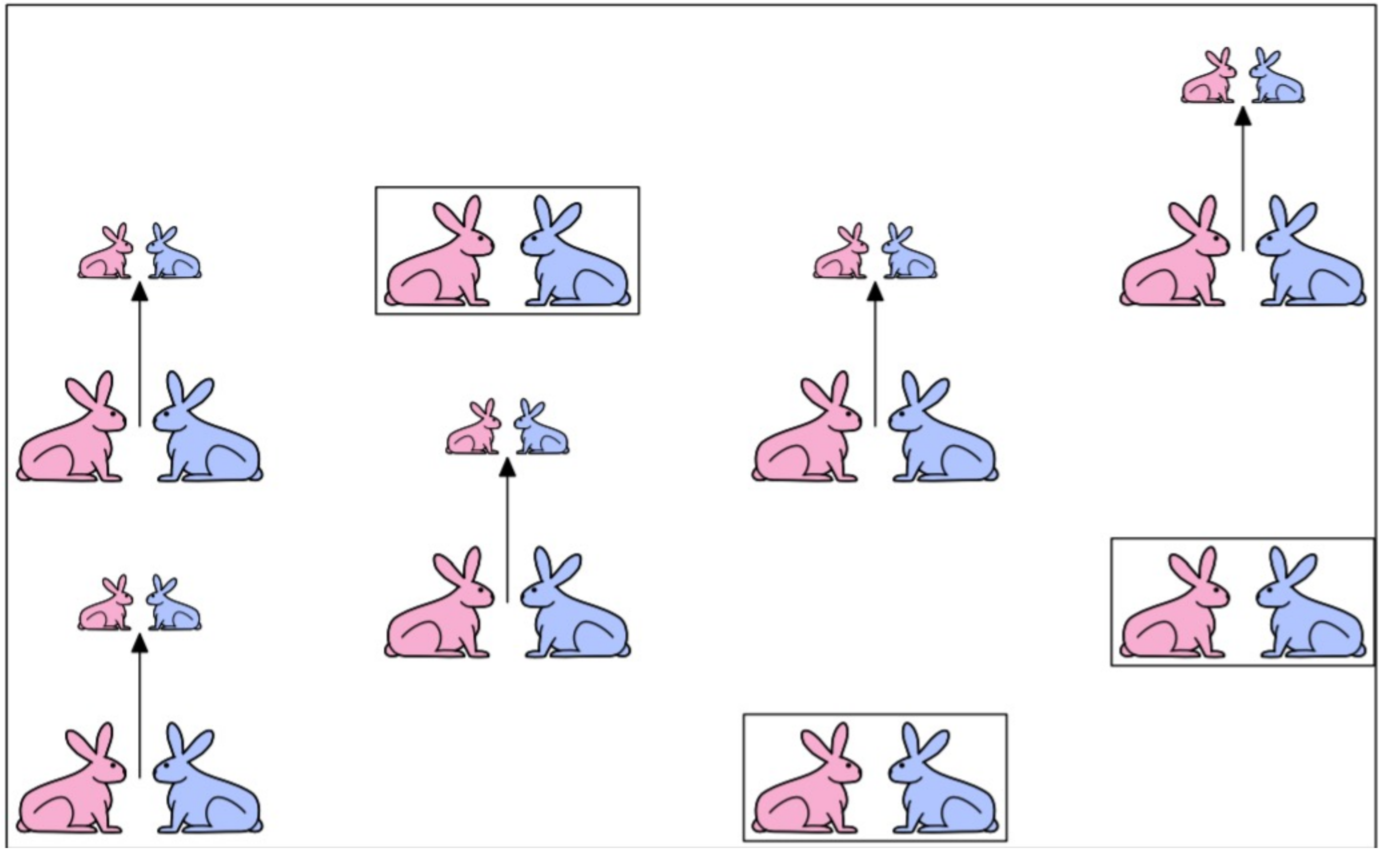












Consolidating the idea

After one month (call it 0) – 1 female

After second month – still 1 female (now pregnant)

After third month – two females, one pregnant, one not

In general, females(n) = females(n-1) + females(n-2)

- Every female alive at month n-2 will produce one female in month n;
- These can be added those alive in month n-1 to get total alive in month n

Idea of code

- Base cases:
  - Females(0) = 1
  - Females(1) = 1
- Recursive case
  - Females(n) = Females(n-1) + Females(n-2)

Recursion with non-numerics : The Problem . . .

- how to check if a string of characters is a palindrome, i.e., reads the same forwards and backwards
  - "Able was I, ere I saw Elba" – attributed to Napoleon
  - "Are we not drawn onward, we few, drawn onward to new era?" – attributed to Anne Michaels

think recursively

- First, convert the string to just characters, by stripping out punctuation, and converting upper case to lower case
- Then
  - Base case: a string of length 0 or 1 is a palindrome
  - Recursive case:
    - If first character matches last character, then is a palindrome if middle section is a palindrome

idea of code

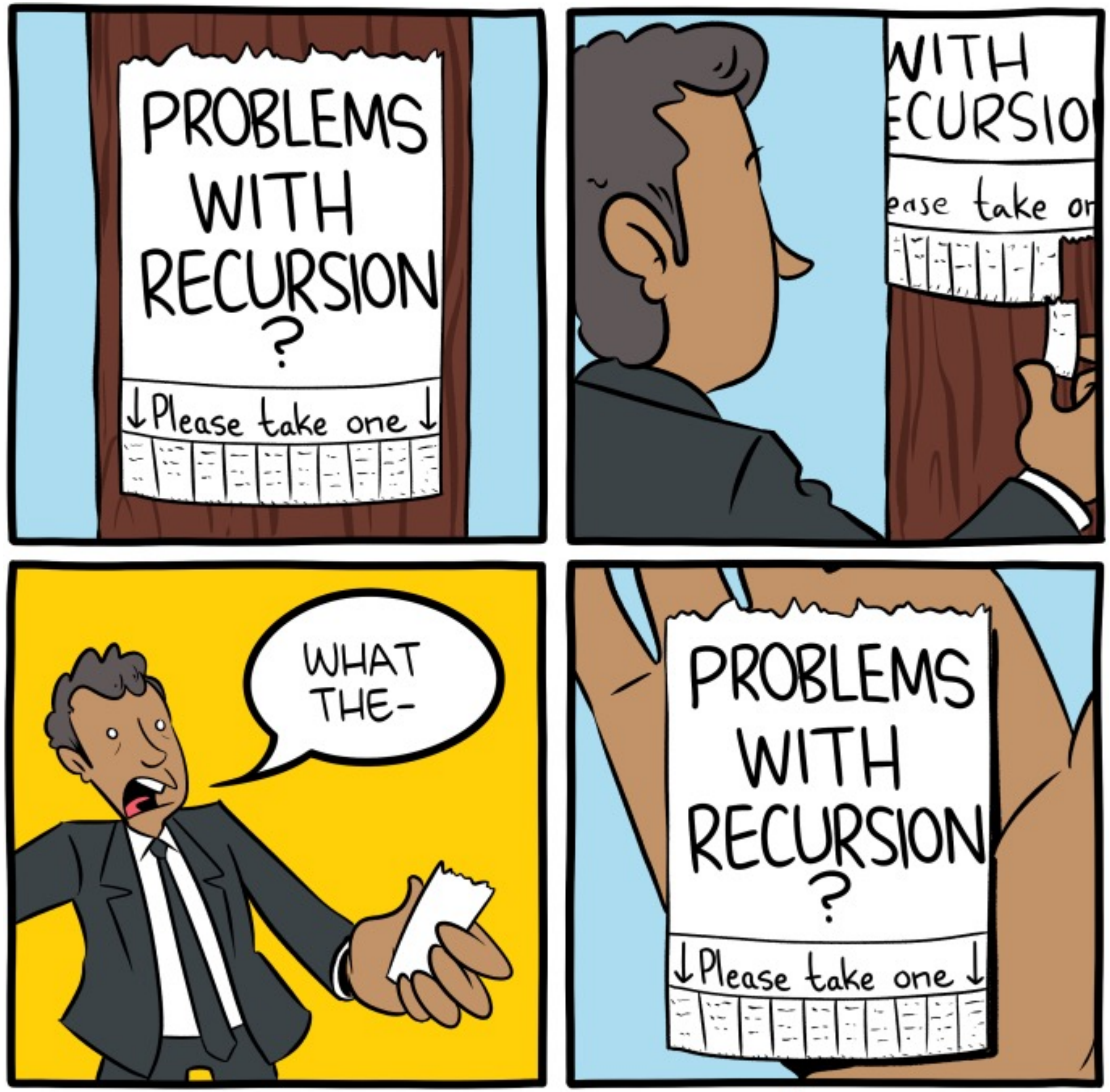
- 'Able was I, ere I saw Elba' → 'ablewasiereisawleba'
- `isPalindrome('ablewasiereisawleba')` is same as
  - `'a' == 'a'` and `isPalindrome('blewasiereisawleb')`

The code

```
def isPalindrome(s):

    def toChars(s):
        s = s.lower()
        ans = ''
        for c in s:
            if c in 'abcdefghijklmnopqrstuvwxyz':
                ans = ans + c
        return ans

    def isPal(s):
        if len(s) <= 1:
            return True
        else:
            return s[0] == s[-1] and isPal(s[1:-1])

    return isPal(toChars(s))
```

takeaway . . .

- an example of a “divide and conquer” algorithm
- solve a hard problem by breaking it into a set of sub-problems such that:
  - sub-problems are easier to solve than the original
  - solutions of the sub-problems can be combined to solve the original


PROBLEMS WITH RECURSION ?
↓Please take one↓
WITH
ECURSIO
ease take on
WHAT THE-
PROBLEMS WITH RECURSION ?
↓Please take one↓

Work to do . . .

- 14 - Recursion
- Programming Assignment 14