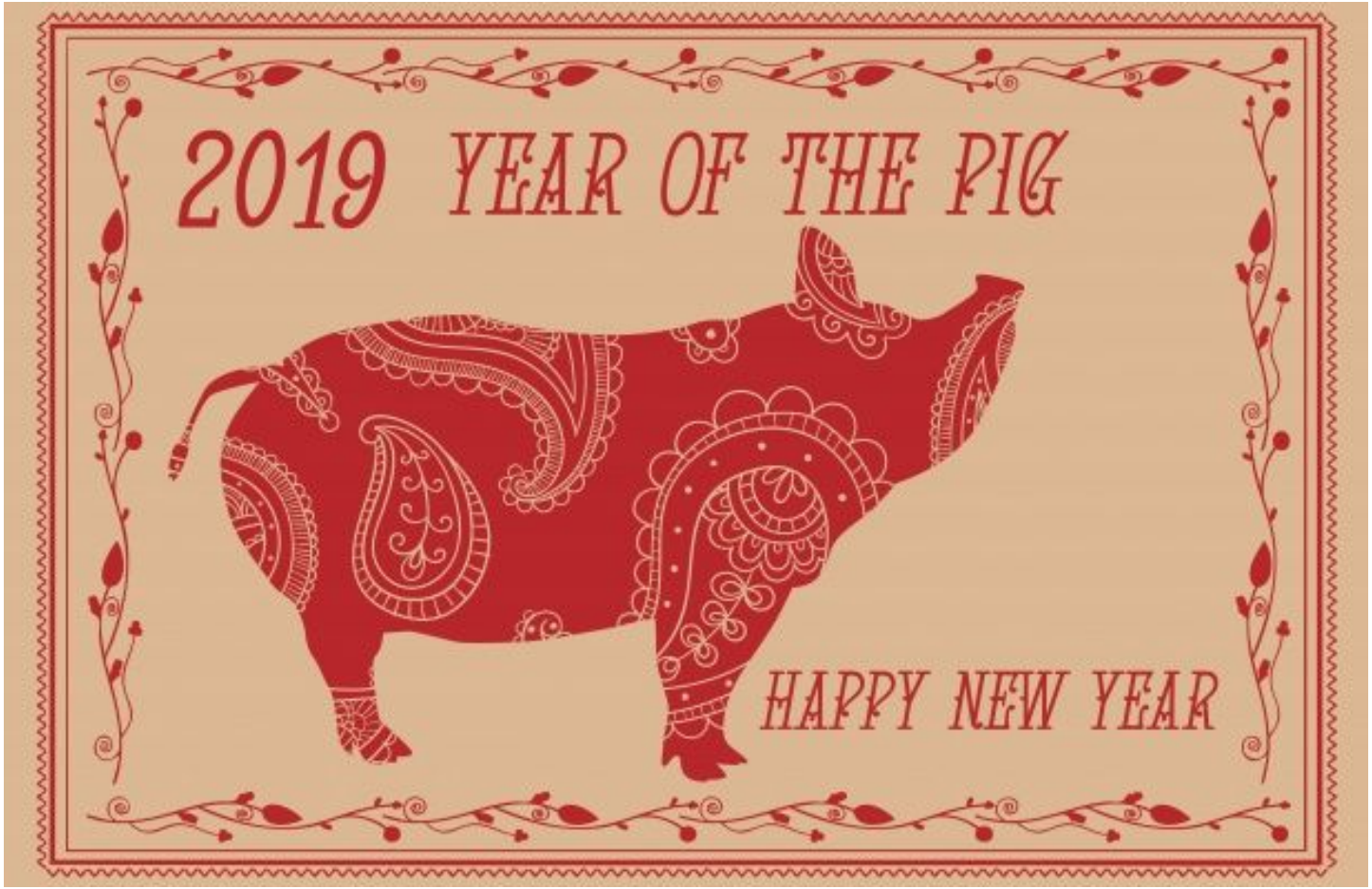
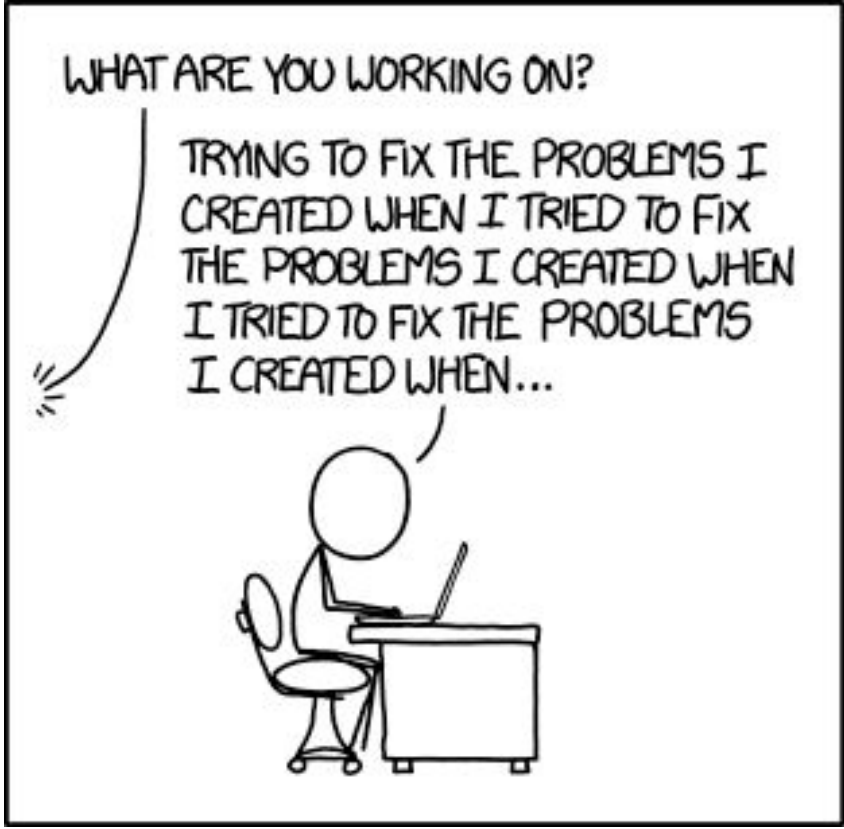
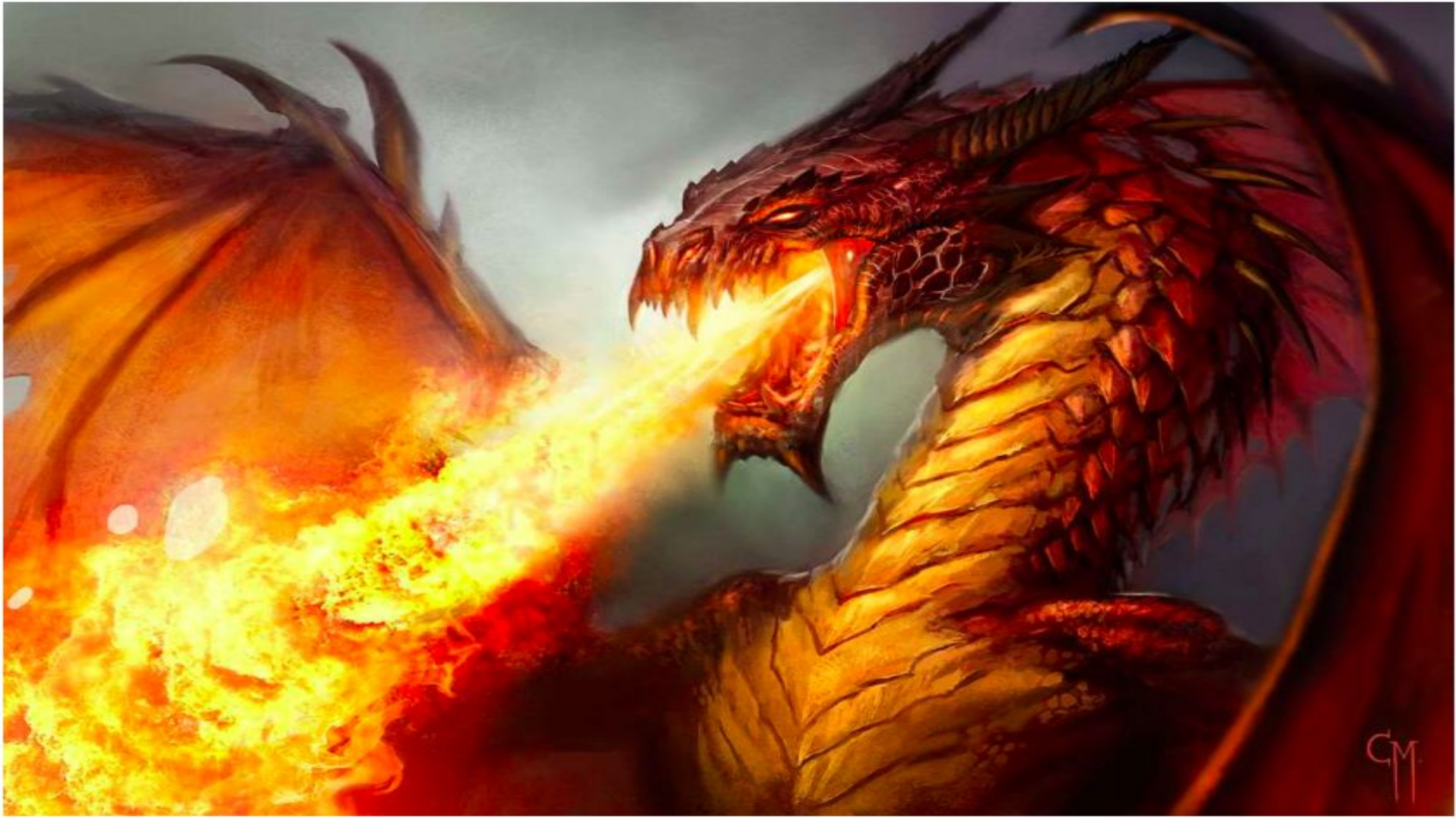

2019 YEAR OF THE PIG
HAPPY NEW YEAR

Lecture 9

Recursion





[3142, 5796, 6550, 8914]

[~~3142~~, 5796, 6550, 8914]

[~~3142~~, ~~5796~~, 6550, 8914]

[~~3142~~, ~~5796~~, ~~6550~~, 8914]

[~~3142~~, ~~5796~~, ~~6550~~, ~~8914~~]

How would you code this? (regular way)

Method that returns TRUE if any element in the array is odd

Method that returns TRUE if any element in the array is odd

```
def anyOdd(lst):
    for num in lst:
        if num % 2 == 1:
            return True
    return False

anyOdd([2, 4, 6, 8])
```

Method that returns TRUE if any element in the array is odd

```
def anyOdd(lst):
    for num in lst:
        if num % 2 == 1:
            return True
    return False

anyOdd([2, 4, 6, 8])
```

We are going to solve this problem using recursion!

What is the Recursion?

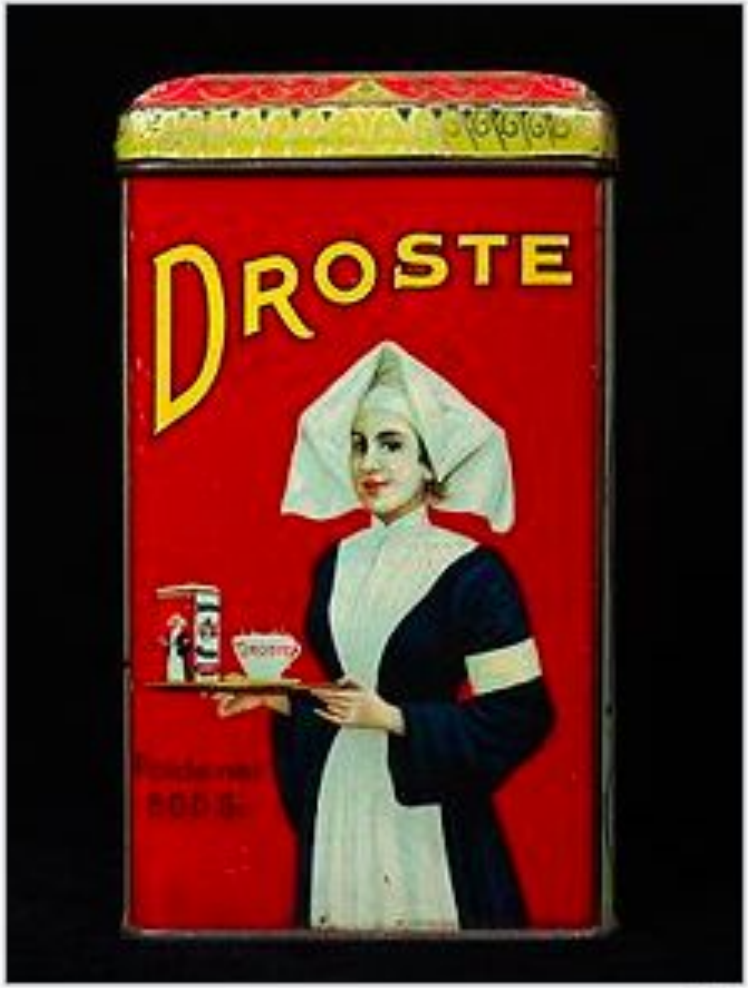
- **Algorithmically**: a way to design solutions to problems by divide-and-conquer method or decrease-and-conquer method
  - Reduce a problem to simpler version of the same problem

Recursion: (1904) ->

In order to understand recursion you must first understand recursion.

GotLines.com

Droste effect ->



https://en.wikipedia.org/wiki/Recursion

Recursion occurs when a thing is defined in terms of itself

Light bulb joke

Q: How many twists does it take to screw in a light bulb?

A: Is it already screwed in? Then zero. If not, then twist it once, ask me again and add 1 to my answer.

What is the Recursion?

- **Algorithmically**: a way to design solutions to problems by divide-and-conquer method or decrease-and-conquer method
  - Reduce a problem to simpler version of the same problem

- **Semantically**: a programming technique where a function calls itself
  - In programming, the goal is NOT to have an infinite recursion
  - Must have at least 1 *base case* that are easy to **solve**
  - Must solve the same problem on some other input with the goal of simplifying the larger problem input.

Recursion: Why?

- **Why do you use it?**
  - Perfect for problems where there is an obvious answer for some small problem and all larger problems build from smaller problems

- There are iterative (**loop based**) solutions for every problem solvable with recursion. Use whichever is simpler
  - Although there may be performance implications of each

Recursion: Step 1

- Solve one step, one part of the problem
  - e.g. is the first number in the array even

- Leave rest of the problem for future steps
  - e.g. the rest of the array

Recursive Functions

Definition: A function is called recursive if the body of that function calls itself, either directly or indirectly.

```
def print_me_again(x):
    if (x == 0):
        return
    print(x)
    print_me_again(x-1)
```

Get to a smaller problem

```
def anyOdd_2(lst):
    if lst[0] % 2 == 1:
        return True
    else:
        smaller = lst[1:]
        return anyOdd_2(smaller)


anyOdd_2([0, 4, 6, 8])

anyOdd_2([])  ->
```

Will this code work?

A: Yes, always

B: Never

C: Sometimes

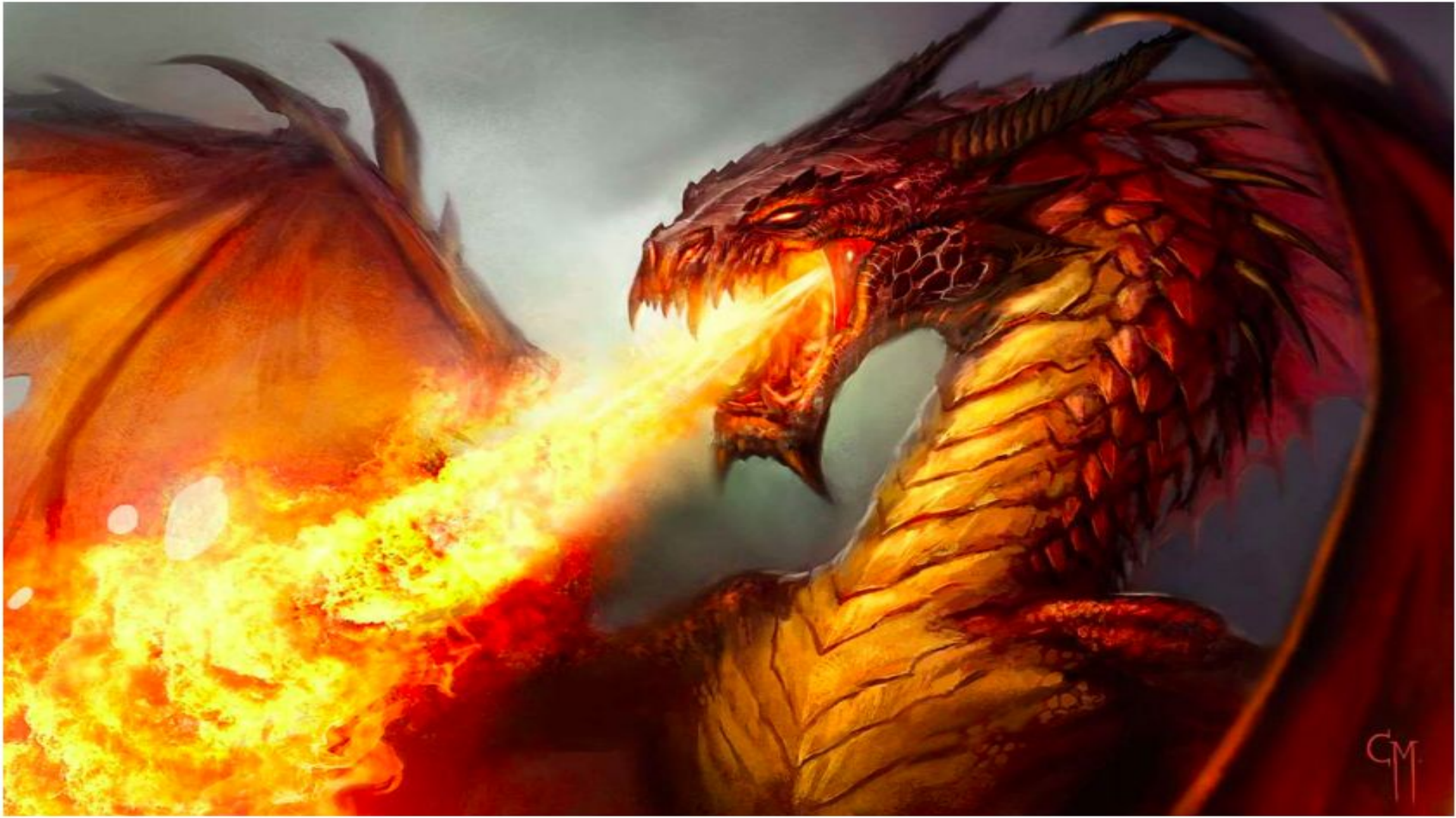
D: I have no idea

Fixed. Trace it

```
def anyOdd_3(lst):
    if (len(lst) == 0 ):
        return False
    elif lst[0] % 2 == 1:
        return True
    else:
        smaller = lst[1:]
        return anyOdd_3(smaller)

anyOdd_3([2, 4, 6, 8])
```

5!



$5 * 4!$

$$5 * 4 * 3!$$

$$5 * 4 * 3 * 2!$$

5 * 4 * 3 * 2 * 1!

$$5 * 4 * 3 * 2 * 1 * 0!$$

$$5 * 4 * 3 * 2 * 1 * 1$$

Question

```
def hello(x):
    print(x)
    hello(x-1)
```

Each number is on a new line ----->

What happens if we call

`hello(0)`

A: Compiler error

B: 0

C: 0 -1 -2 -3 -4 ...

D: 0 -1 -2 -3 -4....until crash

E: None of the above

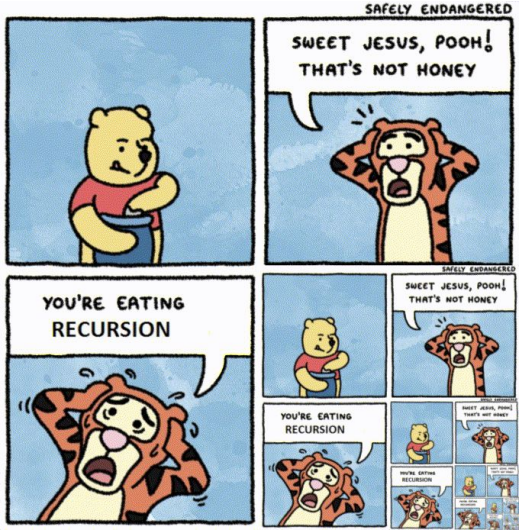
Trace it

```
def hello(x):
    print(x)
    hello(x-1)

hello(0)
```

Recursion: Step 2

- Know when to stop! Known as the base case
  - When the array only has one element (or no elements)
  - Solution to a small problem.



Question (X! = 1 * 2 * 3*...*X)

```
def fact(x):
    if (x == 0):
        _____________
    else:
        ___________________

print(fact(5))
```

Fill in blanks (assume x is not negative)

A: return 0 fact(x)

B: return 1 return x * fact(x-1)

C: return 1 return (x-1) * fact(x-1)

D: return 0 return x * fact(x-1)

E: None of the above

Question

```
def fact(x):
    if (x == 0):
        _____________
    else:
        ____________________

print(fact(5))
```

Fill in blanks (assume x is not negative)

A: return 0 fact(x)

B: return 1 return x * fact(x-1)

C: return 1 return (x-1) * fact(x-1)

D: return 0 return x * fact(x-1)

E: None of the above

Question

```
def fact(x):
    if (x == 0):
        return 1
    else:
        return x * fact(x-1)


print(fact(5))
```

Fill in blanks (assume x is not negative)

A: return 0 fact(x)

B: return 1 return x * fact(x-1)

C: return 1 return (x-1) * fact(x-1)

D: return 0 return x * fact(x-1)

E: None of the above

Steps to design a recursive algorithm

- **Base case:**
  - For small values of n, it can be solved directly
- **Recursive case(s)**
  - Smaller version of the same problem

Steps to design a recursive algorithm

- **Base case:**
  - For small values of n, it can be solved directly
- **Recursive case(s)**
  - Smaller version of the same problem

Algorithmic steps

- Identify the base case and provide a solution to it
- Reduce the problem to smaller version of itself
- Move towards the base case using smaller input versions

Trace it

```
def magic(x):
    if (x>1):
        magic(x-1)
    print(x)

magic(5)
```

What is the output?

A: 5

B: 5 4 3 2 1 #new line for each number

C: 1 2 3 4 5 #new line for each number

D: 1

E: Error or Infinite recursion

Let's practice

Write a recursive method that calculates the product of *a* and *b,* where *a* and *b* are inputs that method. (a, b are positive ints)

Iterative solution:

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

Recursive solution:

```
def mult_rec(a, b):
```

Let's practice + env. diagram

Write a recursive method that calculates the product of a and b where a and b are positive ints.

Iterative solution:

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

Recursive solution:

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

Let's practice

Given a non-negative int n, return the count of the occurrences of 9 as a digit.

- Note that mod (%) by 10 yields the rightmost digit (129 % 10 is 9)
- divide (//) by 10 removes the rightmost digit (129 // 10 is 12)
- No loops

```
def count_9 (number):
    """
    >>> count_9s(919)
    2
    >>> count_9s(7)
    0
    >>> count_9s(129)
    1
    """
    # Base case. Think how many do you need.
```

Let's practice

Given a non-negative int n, return the count of the occurrences of 9 as a digit.

- Note that mod (%) by 10 yields the rightmost digit (129 % 10 is 9)
- divide (//) by 10 removes the rightmost digit (129 // 10 is 12)
- No loops

```
def count_9 (number):
    """
    >>> count_9s(919)
    2
    >>> count_9s(7)
    0
    >>> count_9s(129)
    1
    """
    # Base case. Think how many do you
need.
    if number == 9:
        return 1
    elif number < 9:
        return 0
    else:
```

What is your recursive step?

Let's practice

Given a non-negative int n, return the count of the occurrences of 9 as a digit.

- Note that mod (%) by 10 yields the rightmost digit (129 % 10 is 9)
- divide (//) by 10 removes the rightmost digit (129 // 10 is 12)
- No loops

```
def count_9 (number):
    """
    >>> count_9s(919)
    2
    >>> count_9s(7)
    0
    >>> count_9s(129)
    1
    """
    # Base case. Think how many do you need.
    if number == 9:
        return 1
    elif number < 9:
        return 0
    else:
        rem = number % 10
        smaller = number //10
        if (rem == 9):
            return 1 + count_9s(smaller)
        else:
            return 0 + count_9s(smaller)
```

Given a string, compute recursively the number of x characters in the string

```
def count_x (input):
    """
    >>> count_x("xxhixx")
    4
    >>> count_x("xhixhix")
    3
    >>> count_x("hi")
    0
    """


# base case?
```

Given a string, compute recursively the number of x characters in the string

```
def count_x (input):
    """
    >>> count_x("xxhixx")
    4
    >>> count_x("xhixhix")
    3
    >>> count_x("hi")
    0
    """


# base case?
```

```
if input == '':  # if input is an empty string

        return 0

    else:
```

Given a string, compute recursively the number of x characters in the string

```
def count_x (input):
    """
    >>> count_x("xxhixx")
    4
    >>> count_x("xhixhix")
    3
    >>> count_x("hi")
    0
    """


# base case?
```

```
if input == '':  # if input is an empty string
    return 0
else:
    if input[0] == 'x':
        return 1 + count_x (input[1:])
```

Given a string, compute recursively the number of x characters in the string

```
def count_x (input):
    """
    >>> count_x("xxhixx")
    4
    >>> count_x("xhixhix")
    3
    >>> count_x("hi")
    0
    """


# base case?
```

```
if input == '':  # if input is an empty string
    return 0
else:
    if input[0] == 'x':
        return 1 + count_x (input[1:])
    else:
        return 0 + count_x (input[1:])
```

Check if a given string is a palindrome

```
def is_palindrome (input):
    """
    >>> is_palindrome ('123321')
    True
    >>> is_palindrome('34554')
    False
    >>> is_palindrome('12321')
    True
    """
    # Base case
```

Check if a given string is a palindrome

```
def is_palindrome (input):
    """
    >>> is_palindrome ('123321')
    True
    >>> is_palindrome('34554')
    False
    >>> is_palindrome('12321')
    True
    """
    # Base case
```

```
if len(input) == 0:
        return True
```

Check if a given string is a palindrome

```
def is_palindrome (input):
    """
    >>> is_palindrome ('123321')
    True
    >>> is_palindrome('34554')
    False
    >>> is_palindrome('12321')
    True
    """
    # Base case
```

```
if len(input) == 0:
        return True

if len(input) == 1:
        return True


# recursive step
```

Check if a given string is a palindrome

```
def is_palindrome (input):
    """
    >>> is_palindrome ('123321')
    True
    >>> is_palindrome('34554')
    False
    >>> is_palindrome('12321')
    True
    """
    # Base case
```

```
if len(input) == 0:
        return True

if len(input) == 1:
        return True

# recursive step

else:
    if input[0] == input[-1]:
        return is_palindrome(input[1:-1])
    else:
        return False
```

Check point 1

```
def func(lst):
“””
>>> func([3, 1, 2, 0])
3
1
“””
    if len(lst) == 0:
        return
    else:
        if lst[0] % 2 == 1:
            print(lst[0])
        else:
            func(lst[1:])
```

Correct solution?

A: Yes for all inputs

B: Yes but for some inputs

c: Does not work for any input

Check point 1

```
def func(lst):
“””
>>> func([3, 1, 2, 0])
3
1
“””
    if len(lst) == 0:
        return
    elif lst[0] % 2 == 1:
        print(lst[0])

    func2(lst[1:])
```

Check point 2

```
def func1(n):
    if(n == 1):
        return 0
    else:
        return 1 + func1(n/2)
```

Does it look correct?

A: Yes

B: No, the base case is missing

C: No, the size of the problem does not get smaller

Check point 3

```
def magic(lst):
    if len(lst) == 0:
        return [ ]

    return [lst[-1]] + magic(lst[:-1])


magic([1 ,2 ,3])
```

Purpose of the code?

A: Swap first and last elements in the list

B: Create a palindrome

C: Reverse a given list

D: Reverse the first half of the list

E: To confuse me