CSC 148

Week 7

Sadia ‘Rain’ Sharmin

Classes begin 10 minutes after the hour

This Week's Cool Programmer(s) Feature: *Rahma Ali*

Week 7

Recursion

SUBTOPICS:

- Intro to Recursion
- Tracing Recursive Functions
- Writing Code Using Recursive Patterns
- More Complex Recursion
- Recursion Efficiency

Intro to Recursion


in order to understand recursion,
you must first understand recursion

in order to understand recursion, you must first understand recursion

Recursion means “self reference”. When something refers to itself or describes itself, it is called recursive.

Recursion

Recursive Problem Solving

Solve a problem by using an algorithm that calls itself on a smaller problem

With each call, the problem becomes simpler

At some point, the problem becomes trivial!

Recursion

Recursive Problem Solving

Let's talk about some example scenarios!

Recursion

What is recursion?

"In order to understand recursion ...
you must first understand recursion"

Actually, to understand recursion, one must understand its recursive components ...

Recursion

Recursion types

Depends on how we split the problem

"N-1" approach: handle one entity, then call the recursion for N-1 entities

Divide in 2 or more subproblems: apply recursion for each half, quarter, etc. of the problem

Other ways (more later..)

Recursion

Programmer perspective

Recursion is when a function calls itself directly

Goal:

Calls itself to solve a smaller part of the problem, using the same function/algorithm

In some cases, we need to combine the solution!

Recursion

More formally

Recursion has 2 phases/steps:

1. Base case

   Simplest problem, cannot break it down further

   This is where we stop recursing

2. Recursive decomposition step

   Breaks down the problem into smaller, “similarly-solvable” subproblems

   Must guarantee to eventually get to the base case

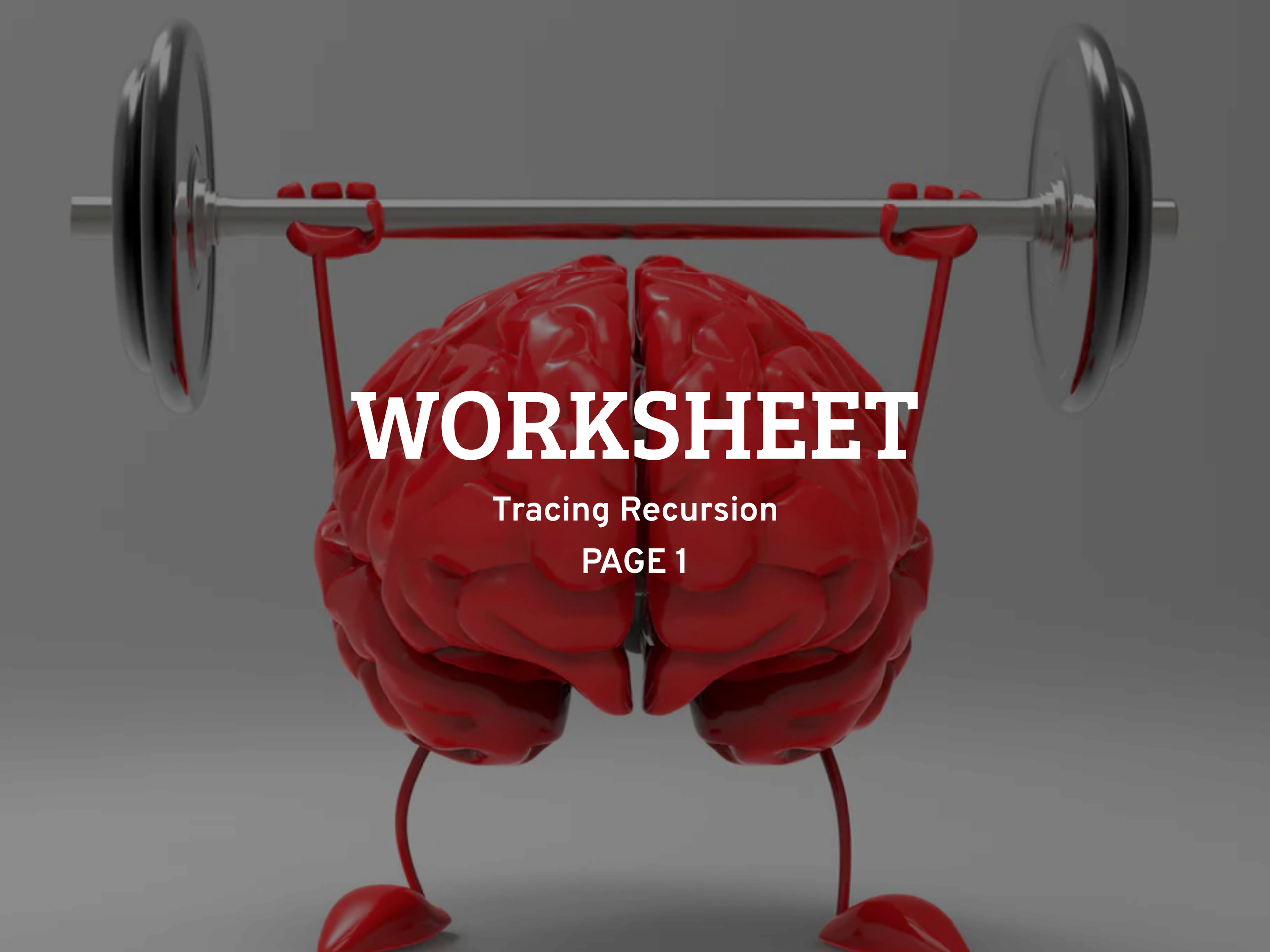
Recursion

Partial tracing practice

Attempting to fully trace recursive code is time-consuming and error prone.

When tracing recursive code, don't trace into recursive calls!

Instead, assume each call is correct, and make sure the rest of the code uses those calls correctly.



WORKSHEET

Tracing Recursion

PAGE 1

Believing in recursion

How can we just *assume* the recursive call works? What it if doesn't?

Let's examine the justification for our confidence.

See confidence_table.pdf

Recursion

Reasoning about correctness

Let P(n) =

"*For any nested list obj of depth n, nested_sum(obj) returns, and returns the sum of the numbers in <obj>.*"

We want to know that ∀n ≥ 0, P(n).

Recursion

Tracing from the smallest case up

For nested_sum, we traced the function and concluded that:

P(0) is true.

P(1) is true as long as P(0) is true.

P(2) is true as long as P(0) and P(1) are true.

. . . and we could have continued on to show that . . .

P(3) is true as long as P(0), P(1), and P(2) are true.

And P(4), P(5), ...

Crucially, we did not trace the “as long as” parts.

(We had already convinced ourselves of them.)

Reasoning more formally

If we show these two things:

P(0) is true.

∀k ≥ 0 ,
P(k+1) is true as long as P(0), … P(k) are all true.

… we can conclude that:

∀n ≥ 0, P(n).

Recursion

Base case(s) in the code

There must be at least one base case.

There may be more than one.

Any call to a recursive method must ultimately reach a base case.

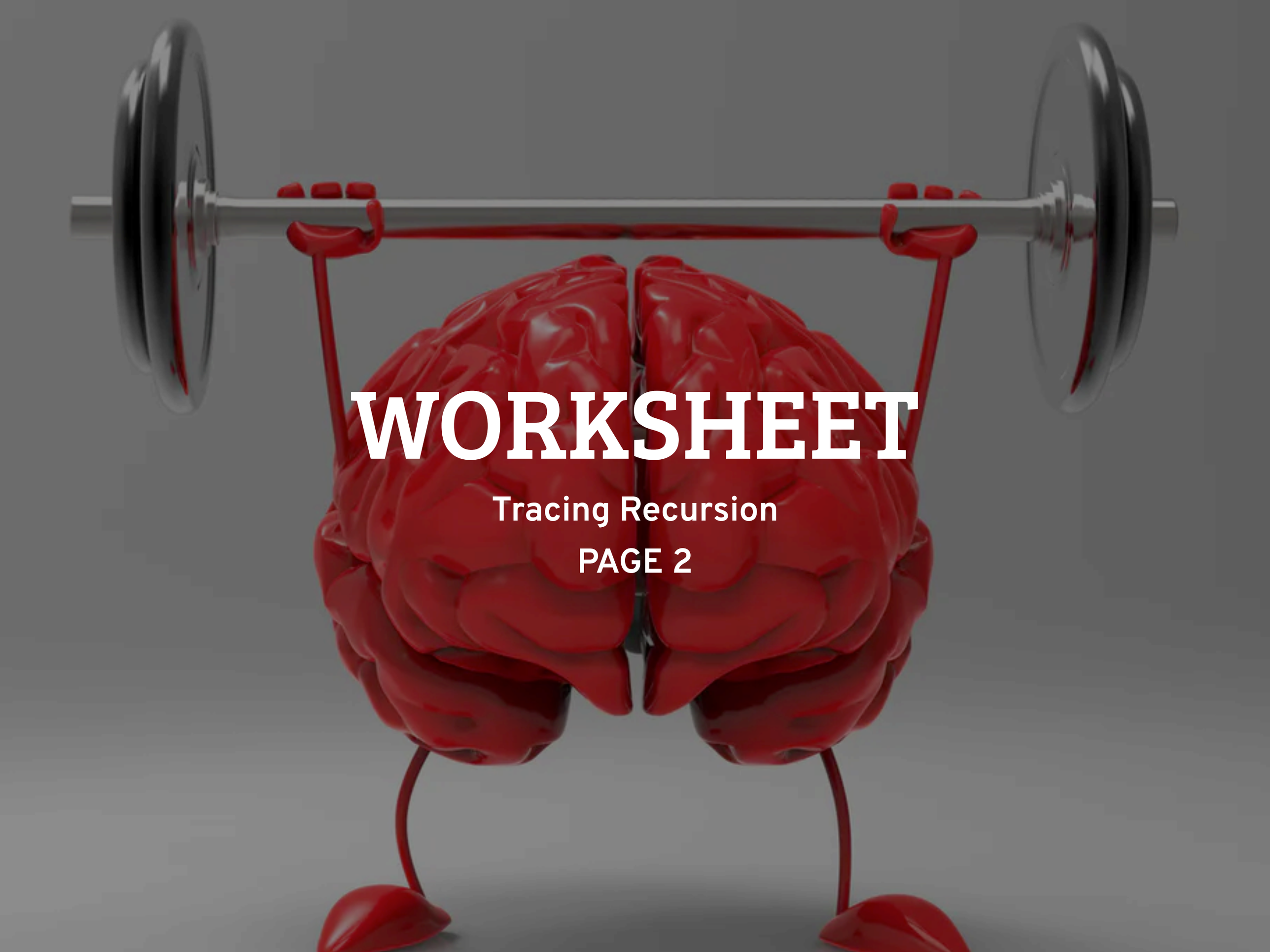
Otherwise, we have “infinite” recursion.

It can’t actually continue indefinitely, because each recursive call needs a stack frame, and we will stop due to running out of memory.

In order to reach a base case:

The problem size must decrease on every recursive call.

Recursive calls must ultimately “connect” with a base case.



WORKSHEET

Tracing Recursion

PAGE 2

Recursive Patterns

Patterns

Note

We have seen before how data structure informs code structure.

A pattern for dealing with linked lists:

```
curr = lst._first
while curr is not None:
    ... curr.item ...
    curr = curr.next
```

Patterns

Note

We have seen before how data structure informs code structure.

A pattern for dealing with lists:

```
for x in list:
    ... x ...
```

Patterns

List of Integers

Let's do an example of summing up a bunch of numbers. We will let the data structure inform our code structure.

If we have a **List[int]** data structure, we can do:

```
def sum_numbers(numlist):
    total = 0
    for num in numlist:
        total += num
    return total
```

Patterns

Nested List

What if we modify the data structure slightly so it's a **List[List[int]]** – i.e. a list containing sublists of integers?

For example, something like:

```
[[1, 2, 3], [4, 5, 6]]
```

and we want to return a sum of all the elements.

Patterns

Nested List

What if we modify the data structure slightly so it's a **List[List[int]]** – i.e. a list containing sublists of integers?

For example, something like

```
[[1, 2, 3], [4, 5, 6]]
```

Would the code we have before need to change?

```
def sum_numbers(numlist):
    total = 0
    for num in numlist:
        total += num
    return total
```

Nested List

What if we modify the data structure slightly so it's a list containing sublists of integers?

Our code changes a bit too:

```
def sum_numbers(numlist):
    total = 0
    for sublist in numlist:
        for num in sublist:
            total += num
    return total
```

Notice that the structure of the code follows the structure of the data we are dealing with — *the list is two levels deep, and the loops are nested two levels deep.*

Patterns

Nested List

Let's add one more difference to the data structure.

What if our list could have either nested lists or integers as its elements **List[Union[int, List[int]]** – how would we modify our code to deal with this so it still adds all the integers up?

For example, something like:

```
[2, [1, 3], 8, [4, 5, 6]]
```

Nested List

What if our list could have either nested lists or integers as its elements **List[Union[int, List[int]]** – how would we modify our code to deal with this so it still adds all the integers up?

For example, something like:

```
[2, [1, 3], 8, [4, 5, 6]]
```

Since integers and lists are very different, we will handle them separately.

Patterns

Nested List

For **List[Union[int, List[int]]** data structure –

Since integers and lists are very different, we will handle them separately.

```
def sum_numbers(numlist: List[Union[int, List[int]]) -> int:
    total = 0
    for element in numlist:
        if isinstance(element, int):
            total += element
        else: # the element is a list of ints
            for num in element:
                total += num
    return total
```

Nested List

Let's complicate things further!

What if our list was three levels deep? For example

```
[[1, [2, 3], 4], [[5], 6], 7]
```

Would this code still work?

```
def sum_numbers(numlist: List[Union[int, List[int]]]) -> int:
total = 0
for element in numlist:
    if isinstance(element, int):
        total += element
    else: # the element is a list of ints
        for num in element:
            total += num
return total
```

Nested List

Let's complicate things further!

What if our list was three levels deep? For example

```
[[1, [2, 3], 4], [[5], 6], 7]
```

Would this code still work? No! We would need to add in yet another nested loop.

If it was four levels deep, again we would need to add another nested loop to deal with that.

And so on! What a pain... and our code would get quite ugly.

Patterns

Arbitrarily Nested List

What if we don't even know how deep the list goes? Let's say we have a list that is nested to some arbitrary depth. All we know is:

A nested list of integers is a list in which every element is either:

- an integer
- yet another nested list of integers

Patterns

Recursive Structures

A nested list of integers is a list in which every element is either:

an integer, or

yet another nested list of integers

This data structure is recursive; it is included in its own definition. Inside of nested lists of integers can be smaller nested lists of integers, inside of which can be smaller ones still, and so on.

Recursion

Recursive Structure

Recursive structures lend themselves well to recursive problem solving.

We already wrote the code for a function which sums up all the elements in a nested list:

```
def sum_numbers(numlist: List[Union[int, List[int]]) -> int:
    total = 0
    for element in numlist:
        if isinstance(element, int):
            total += element
        else: # the element is a list of ints
            for num in element:
                total += num
    return total
```

Recursion

Recursive Structure

Since we now know that the sublist could itself be another similarly structured nested list, we use this function itself as a helper!

We're believing in recursion here, and trusting that our function works as it is supposed to.

```
def sum_numbers(numlist) -> int:
    total = 0
    for element in numlist:
        if isinstance(element, int):
            total += element
        else: # the element is a list of ints
             total += sum_numbers(element)
    return total
```

WORKSHEET

Writing recursive functions

PAGE 1

A common error: missing `return`

```
def nested_list_contains(obj, item) -> bool:
    if isinstance(obj, int):
        return obj == item
    else:
        for sublist in obj:
            if nested_list_contains(sublist, item):
                nested_list_contains(sublist, item)
        return False
```

A common error: missing `return`

```
def nested_list_contains(obj, item) -> bool:
    if isinstance(obj, int):
        return obj == item
    else:
        for sublist in obj:
            if nested_list_contains(sublist, item):
                nested_list_contains(sublist, item)
        return False
```

A common error: missing `return`

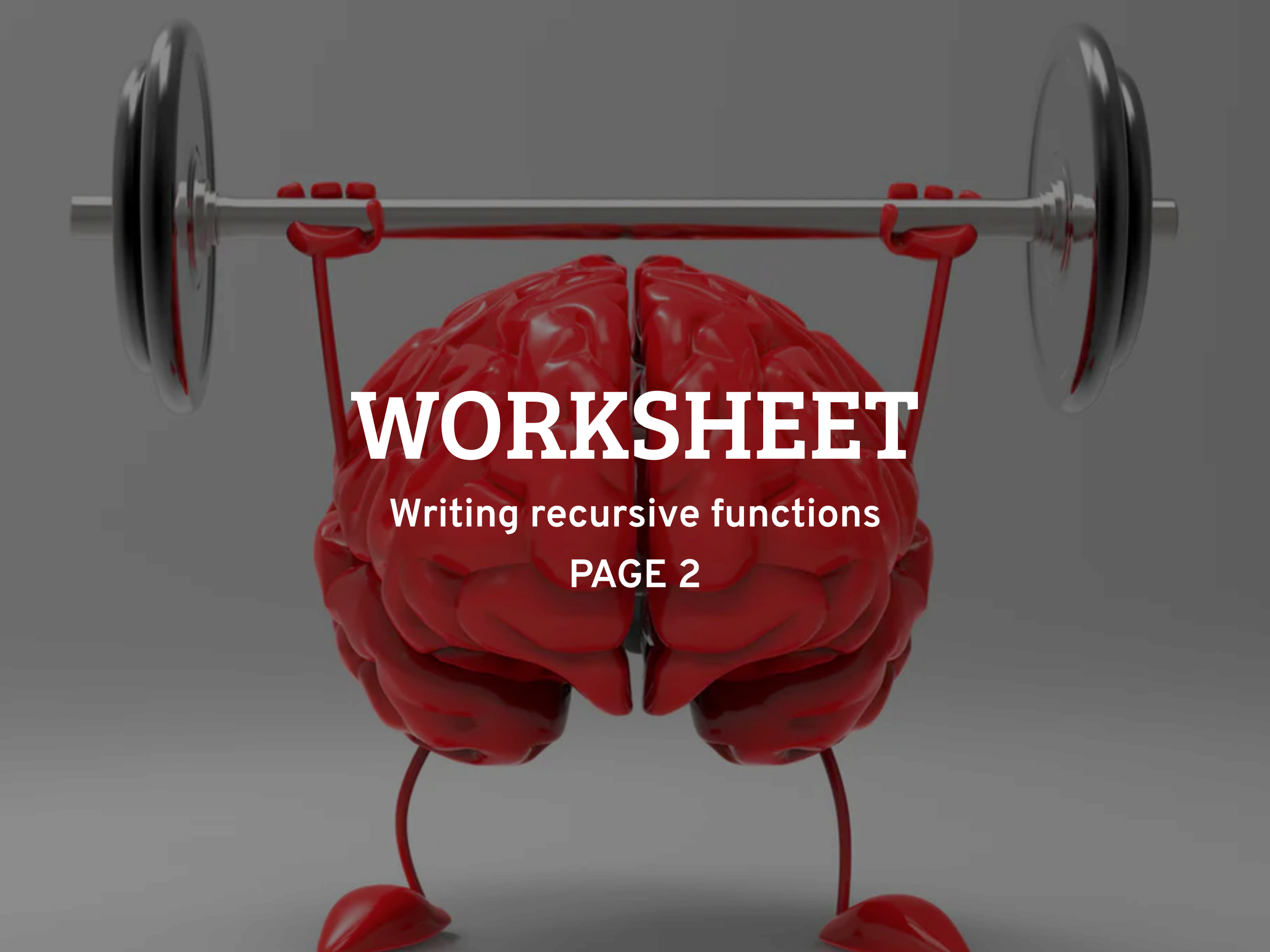
```
def nested_list_contains(obj, item) -> bool:
    if isinstance(obj, int):
        return obj == item
    else:
        for sublist in obj:
            if nested_list_contains(sublist, item):
                True
        return False
```

A common error: missing `return`

A **`return`** statement exits from one function call.

When writing a recursive function that should return something, both the base case and recursive step *typically* must have a `return`!

More generally, if a function returns something, then every execution path through the function must have a **`return`**!



WORKSHEET

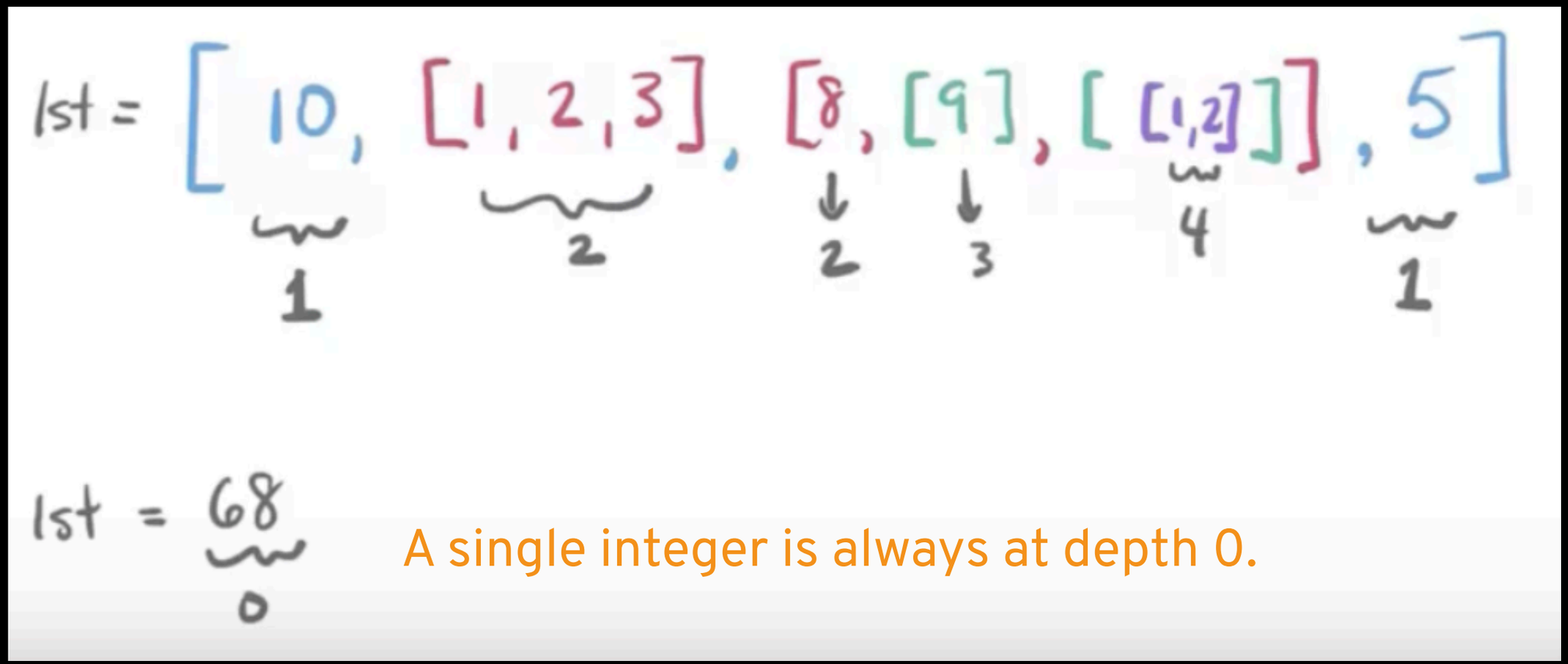
Writing recursive functions

PAGE 2

Recursion

`first_at_depth`

What is the depth of an object within a nested list?



Recursion

`first_at_depth` – base case

A single integer is always at depth 0.

```
>>> first_at_depth(100, 0)
100
>>> first_at_depth(100, 3) is None
True
```

Recursion

`first_at_depth` — recursive case

```
>>> first_at_depth([10, [[20]], [30, 40]], 2)
30
```

sublist	depth	first_at_depth(sublist, depth)
10		
[[20]]		
[30, 40]		

Recursion

`first_at_depth` — multiple base cases!

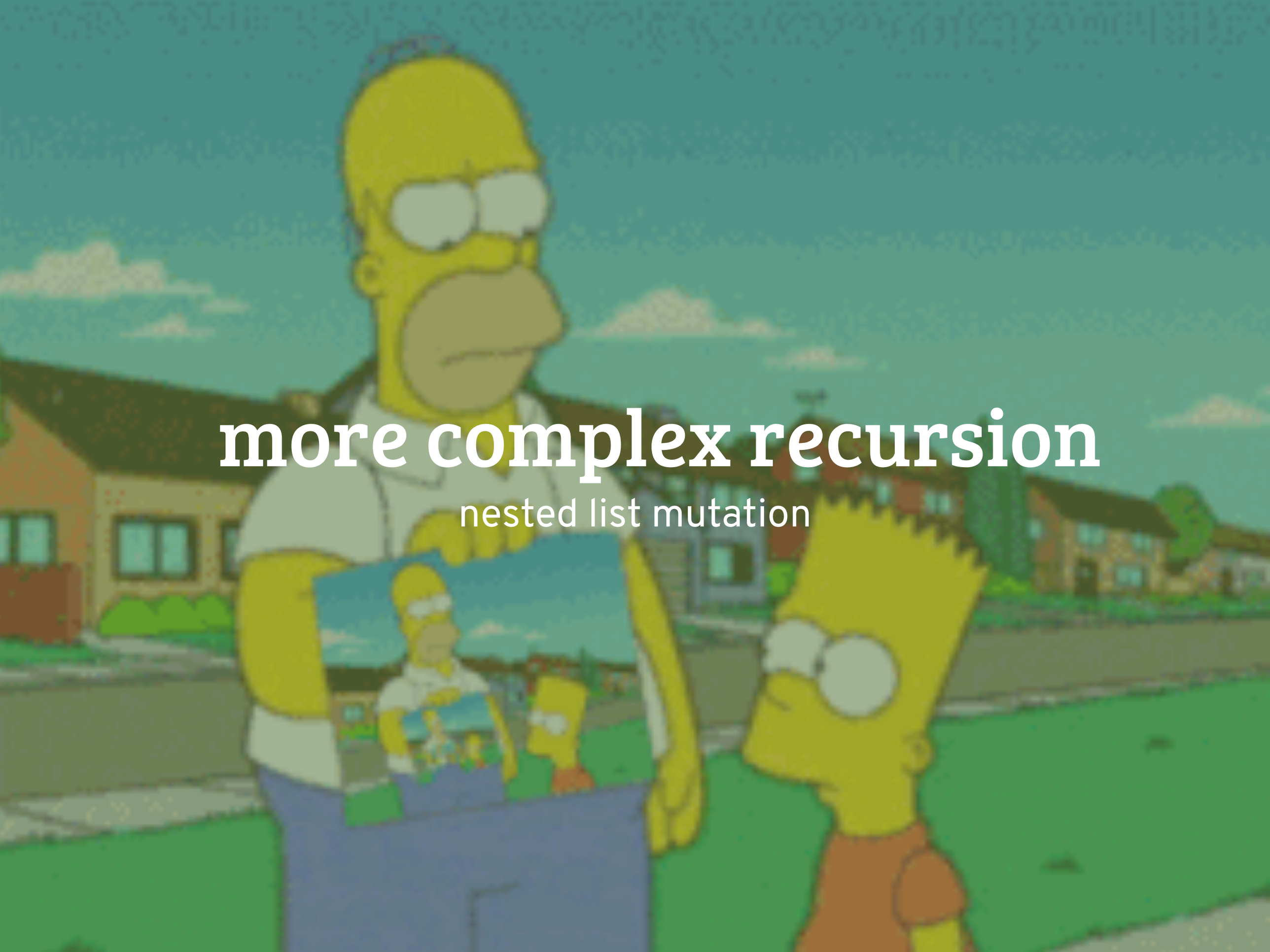
```
first_at_depth(obj, d)
    -> first_at_depth(sublist, d - 1)
```

We are actually recursing on both `obj` and `d`.

Can't recurse when:

```
isinstance(obj, int)
```

```
d == 0
```



more complex recursion

nested list mutation

WORKSHEET

Nested List Mutation

Recursion Efficiency

Recursion

Recursion and redundancy

Remember recursion:

Calculating Fibonacci numbers

if n < 2, fib(n) = 1

fib(n) = fib(n-1) + fib(n-2)

Write a recursive program for this..

```
def fib(n: int) -> int:
    """
    Returns the n-th fibonacci number.
    """
    pass
```

Recursion

Recursion and redundancy

Remember recursion:

Calculating Fibonacci numbers

if n < 2, fib(n) = 1

fib(n) = fib(n-1) + fib(n-2)

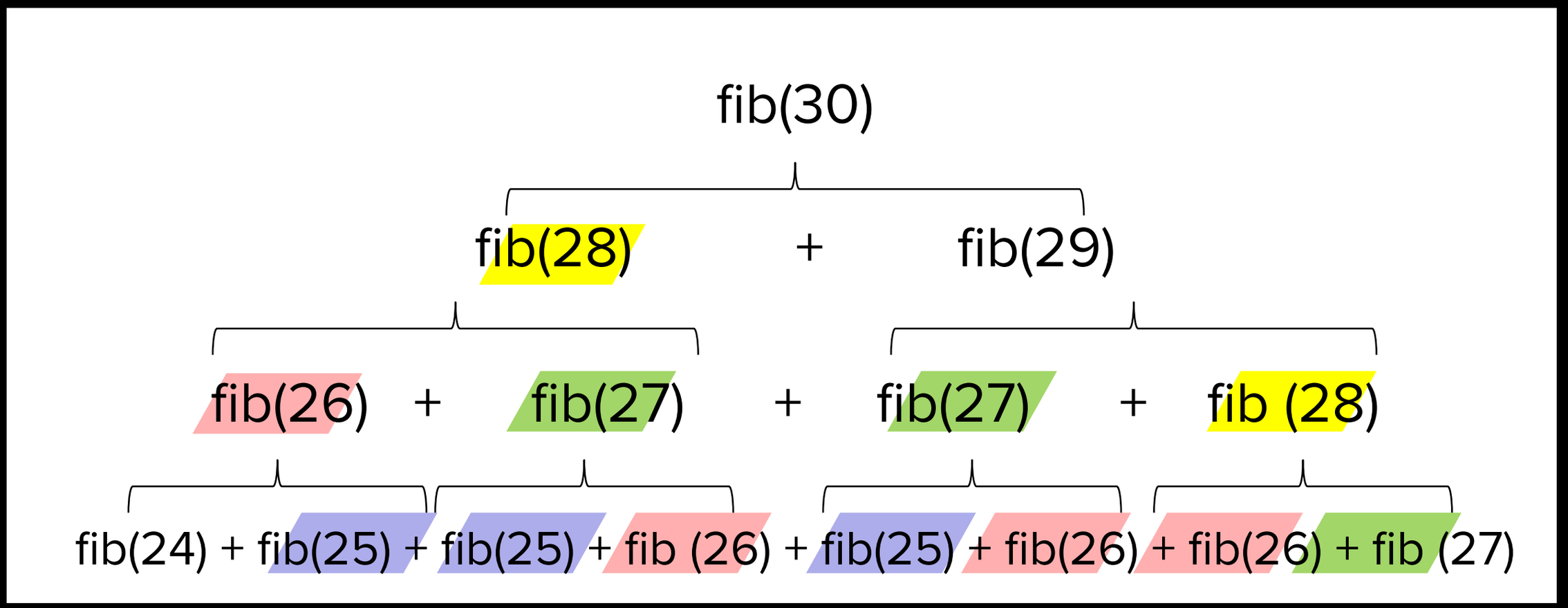
Write a recursive program for this..

```
def fib(n: int) -> int:
    """
    Returns the n-th fibonacci number.
    """
    if n < 2:
        return 1
    else:
        return fib(n-1) + fib(n-2)
```

Redundancy

Unnecessary repeated calculations => inefficient!

Let's expand the recursion: fib(n) = fib(n-1) + fib(n-2)



How could we avoid calculating items we already calculated?

Recursion

Solution? Memoize

Keep track of already calculated values

```
def fib_memo(n: int,  seen: Dict[int, int]) -> int:
    """
    Returns the <n>-th fibonacci number, reasonably quickly, without
    redundancy.
    Parameter <seen> is a dictionary of already-seen results
    """
    if n not in seen:
        if n < 2:
            seen[n] = n
        else:
            seen[n] = fib_memo(n-2,  seen)  +  fib_memo(n-1,  seen)

    return seen[n]
```

Running out of stack space

Some programming languages have better support for recursion than others; python may run out of space on its stack for recursive function calls ...

For example, recursively traversing a **very** long list ...

Recursion

Recursive vs iterative

Any recursive function can be written iteratively

- May need to use a stack too, potentially

Recursive functions are not more efficient than the iterative equivalent

- Could be the same, with compiler support..

Why ever use recursion then?

- If the nature of the problem is recursive, writing it iteratively can be

a) more time consuming, and/or

b) less readable

Recursion

In conclusion

Recursive functions are not always more efficient than their iterative equivalent

But .. Recursion is a powerful technique for naturally recursive problems