RECURSION ON NON-NUMERICS

(download slides and .py files to follow along)

6.100L Lecture 16

Ana Bell

REVIEW OF RECURSION FROM LAST LECTURE, WITH AN EXAMPLE

- Fibonacci numbers (circa 1202)
- Leonardo of Pisa (aka Fibonacci) modeled rabbits mating (under certain assumptions) as a Fibonacci sequence
  - 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) each month from its second month on
- females(n) = females(n-1) + females(n-2)

Females alive in month n-1

Every female alive at month n-2 will produce one female in month n

Month	Females
1	1
2	1
3	2
4	3
5	5
6	8
7	13

FIBONACCI

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

FIBONACCI RECURSIVE CODE (MULTIPLE BASE CASES)

```
def fib(x):
    if x == 1 or x == 2:
        return 1
    else:
        return fib(x-1) + fib(x-2)
```

Two base cases, because have two recursive calls in general case

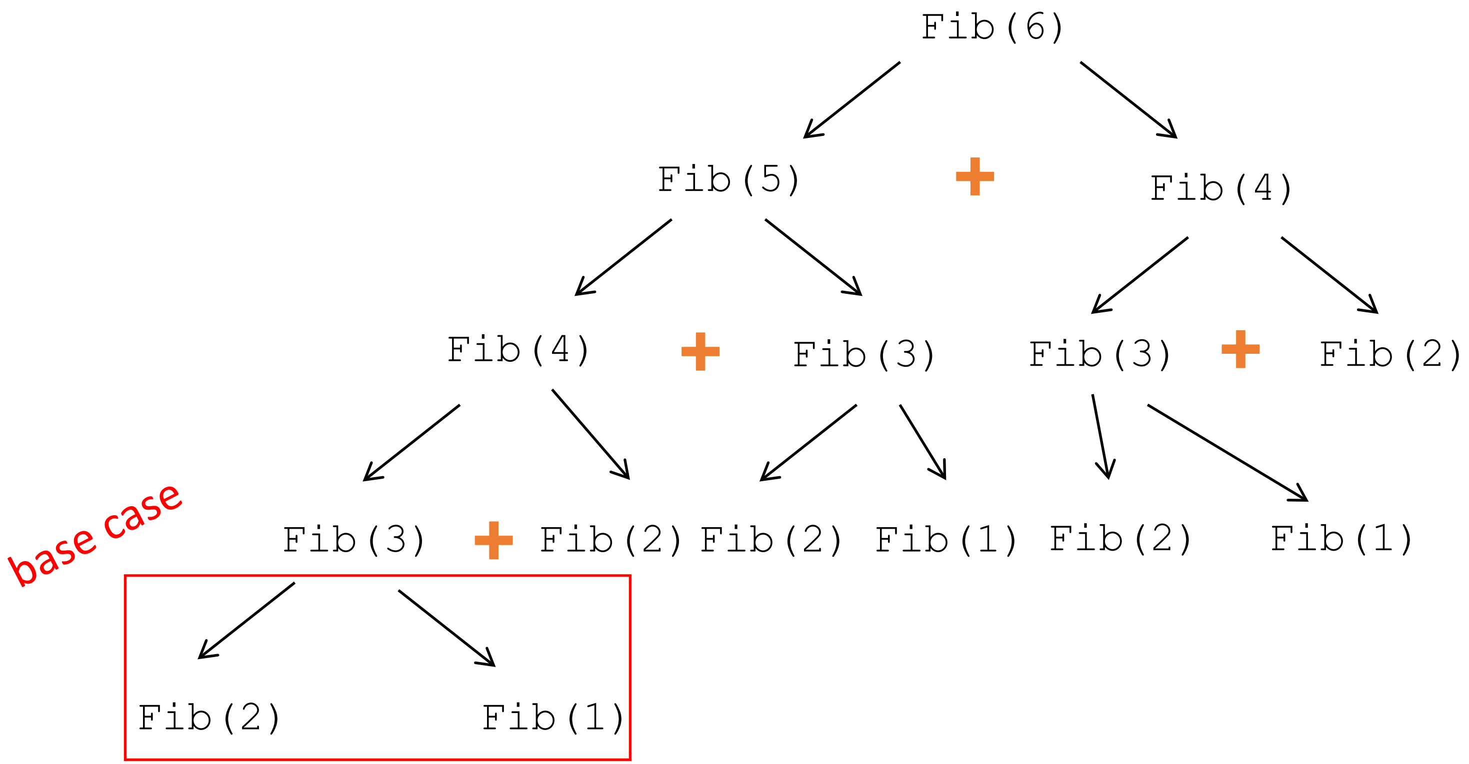
Two base cases

- Calls itself twice
- But! It has to go to the **base case of the first fib call** before completing the second fib call

HIGH-LEVEL VIEW OF FIBONACCI with RECURSION

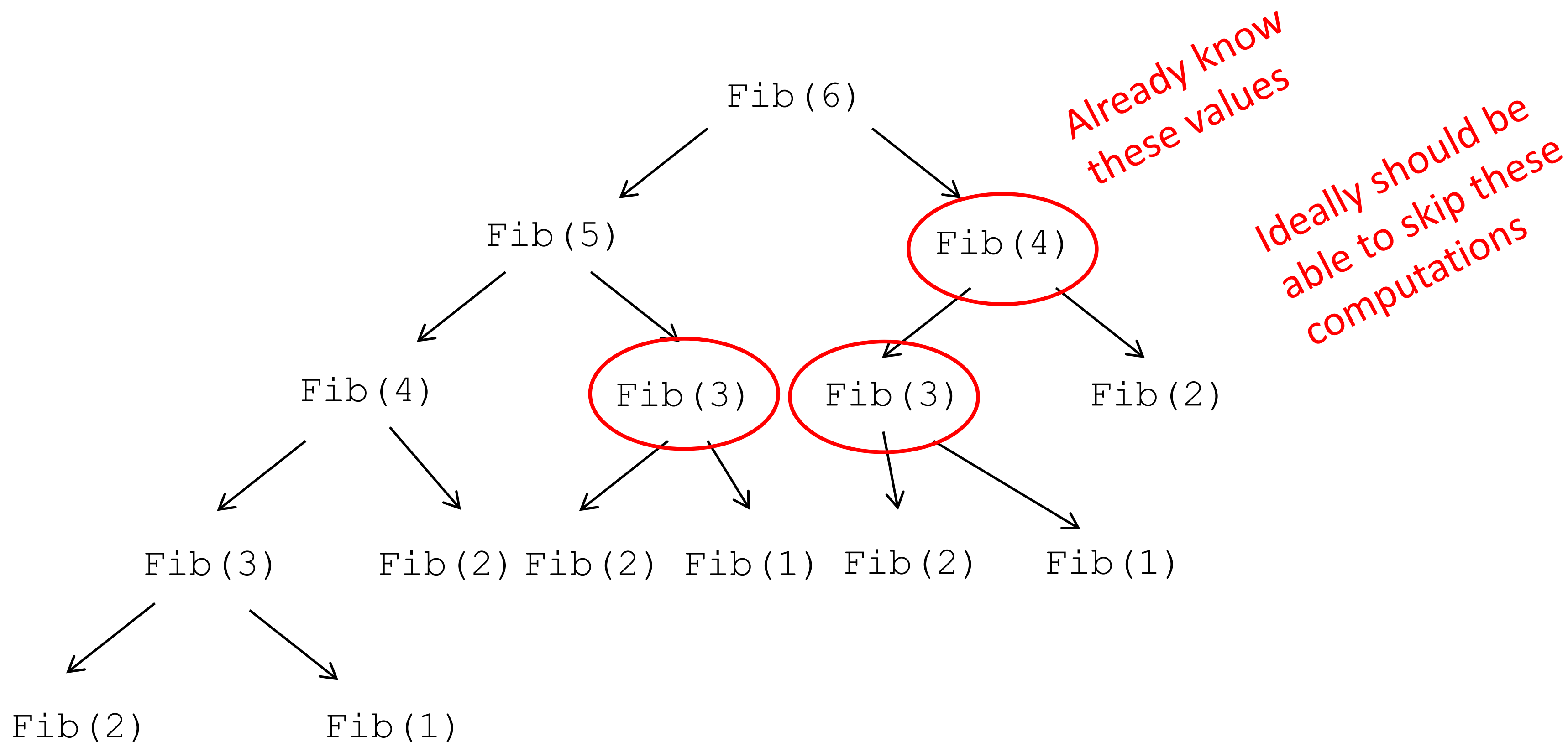
[PYTHON TUTOR LINK]

```
def fib(x):
        if x == 1 or x == 2:
         return 1
    else:
         return fib(x-1) + fib(x-2)
```



INEFFICIENT FIBONACCI

`fib(x) = fib(x-1) + fib(x-2)`



- **Recalculating** the same values many times!
- Could keep **track** of already calculated values

FIBONACCI WITH **MEMO**IZATION

Python Tutor LINK

```
def fib_efficient(n, d):
    if n in d:
        return d[n]
    else:
        ans = fib_efficient(n-1, d) + fib_efficient(n-2, d)
        d[n] = ans
        return ans

d = {1:1, 2:1}
print(fib_efficient(6, d))
```

Before recursively finding Fibonacci, check if we already have the value

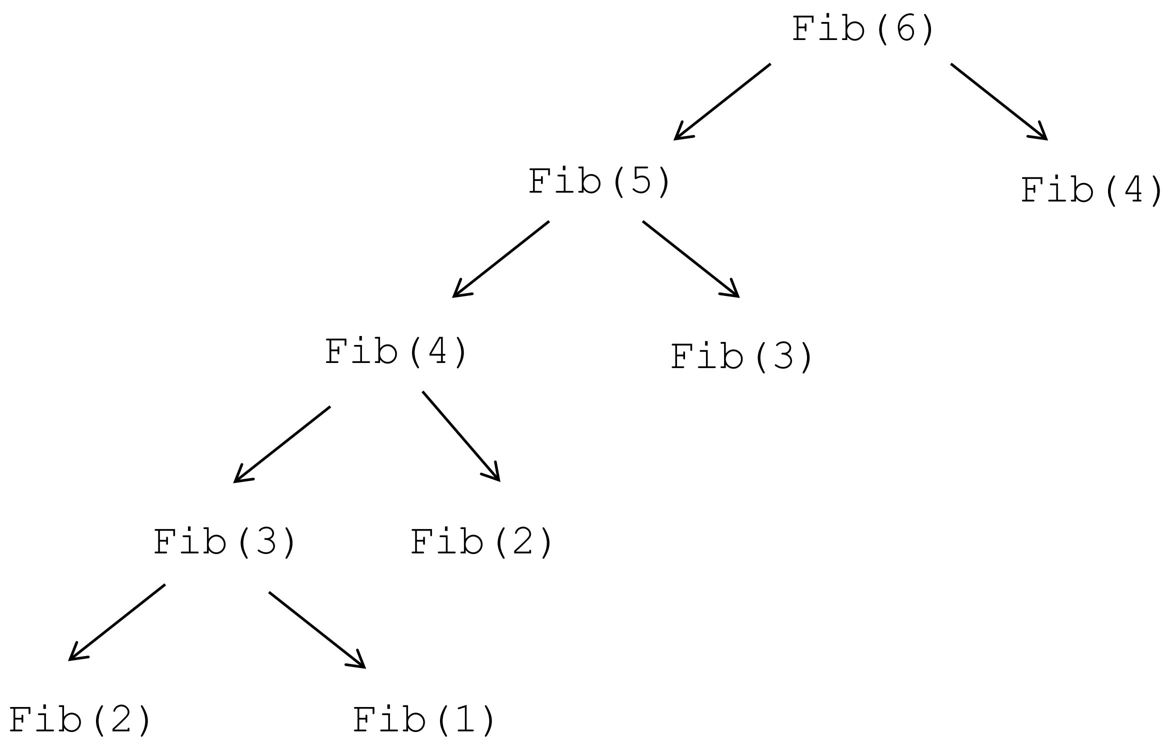
Calculate it

Add to dictionary

Initialize dictionary with base cases

- Do a **lookup first** in case already calculated the value
- **Modify dictionary** as progress through function calls

EFFICIENT FIBONACCI CHECKS the DICT FIRST



n	fib(n)
1	1
2	1
3	2
4	3
5	5
6	8

- **No more recalculating**, just check the dict before calculating!
- Add to the dict so we can **look it up next time** we see it

EFFICIENCY GAINS

- Calling `fib(34)` results in **11,405,773** recursive calls to the procedure
- Calling `fib_efficient(34)` results in **65** recursive calls to the procedure
- Using dictionaries to capture intermediate results can be very efficient
- But note that this only works **for procedures without side effects** (i.e., the procedure will always produce the same result for a specific argument independent of any other computations between calls)

A MORE PRACTICAL EXAMPLE

WHAT ARE ALL THE WAYS YOU CAN MAKE A SCORE OF x IN BASKETBALL?

```
def score_count(x):
    """ Returns all the ways to make a score  of x by adding
    1, 2, and/or 3 together. Order doesn't matter. """
    if x == 1:
        return 1
    elif x == 2:
        return 2
    elif x == 3:
        return 3
```

Score of 1 made with:
1+0

Score of 2 made with:
1+1 or 2+0

Score of 3 made with:
1+1+1 or 2+1 or 3+0

In basketball you can make a basket worth 1, 2, or 3 points

- Base cases: **3 of them**!
    - You can make a score of 1 with 1+0 (that's 1 way)
    - You can make a score of 2 with 1+1 or 2+0 (that's 2 ways)
    - You can make a score of 3 with 1+1+1 or 2+1 or 3+0 (that's 3 ways)

A MORE PRACTICAL EXAMPLE: PYTHON TUTOR LINK

WHAT ARE ALL THE WAYS YOU CAN MAKE A SCORE OF x IN BASKETBALL?

```
def score_count(x):
    """ Returns all the ways to make a score  of x by adding
    1, 2, and/or 3 together. Order doesn't matter. """
    if x == 1:
        return 1
    elif x == 2:
        return 2
    elif x == 3:
        return 3
    else:
        return score_count(x-1)+score_count(x-2)+score_count(x-3)
```

All ways to make a score of x-1

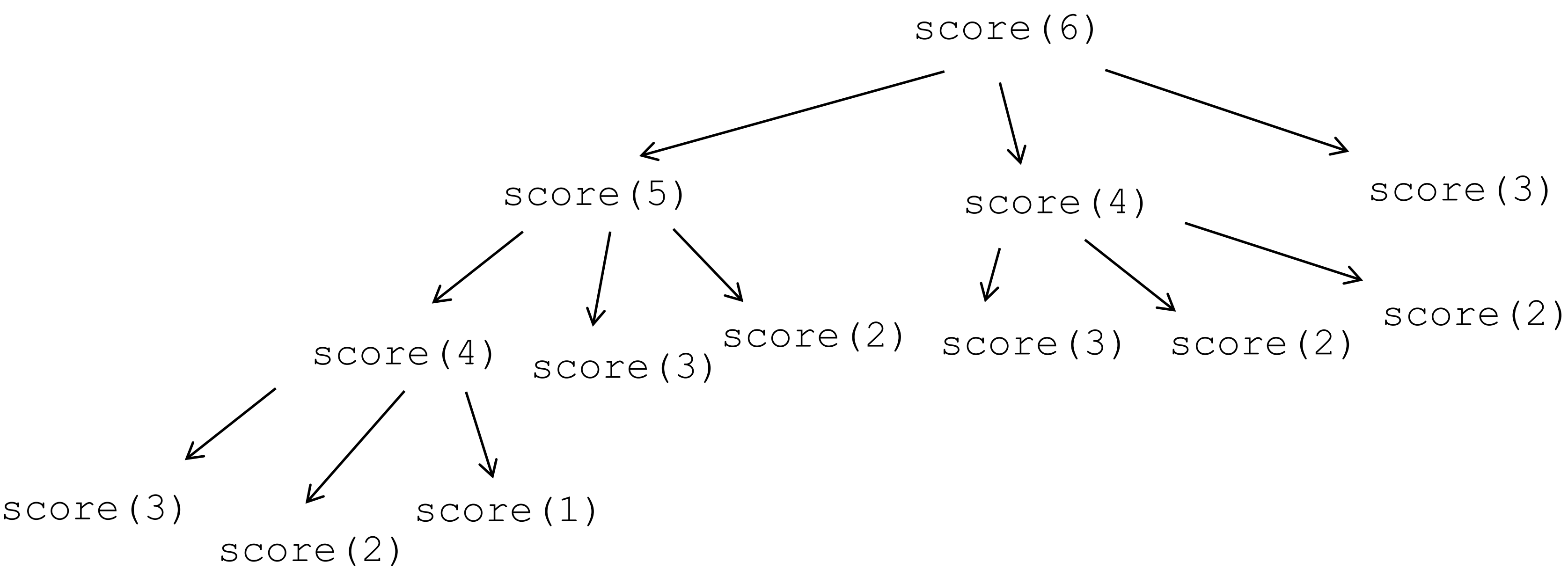
All ways to make a score of x-2

All ways to make a score of x-3

- Recursive step: Let **future function calls do the work** down until base cases
  - Ways to make a score of x means you could have made:
    a score of (x-1) or a score of (x-2) or a score of (x-3)
  - If you make a score of x-1 you can just add 1 to it to make the score of x.
  - If you make a score of x-2 you can just add 2 to it to make the score of x.
  - If you make a score of x-3 you can just add 3 to it to make the score of x.

HIGH-LEVEL VIEW of score_count

```
def score_count(x):
  if x == 1:
    return 1
  elif x == 2:
    return 2
  elif x == 3:
    return 3
  else:
    return score_count(x-1)+score_count(x-2)+score_count(x-3)
```

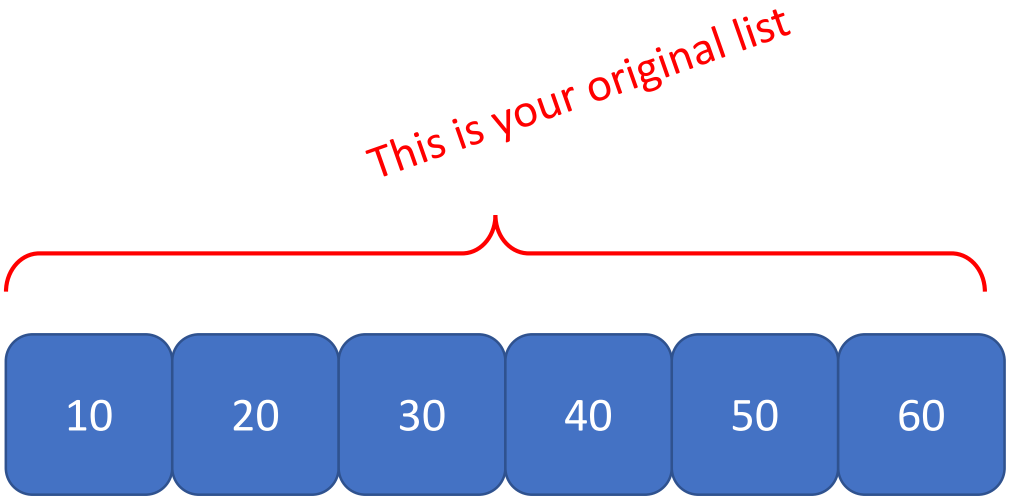


SUM of LIST ELEMENTS

LISTS ARE NATURALLY RECURSIVE

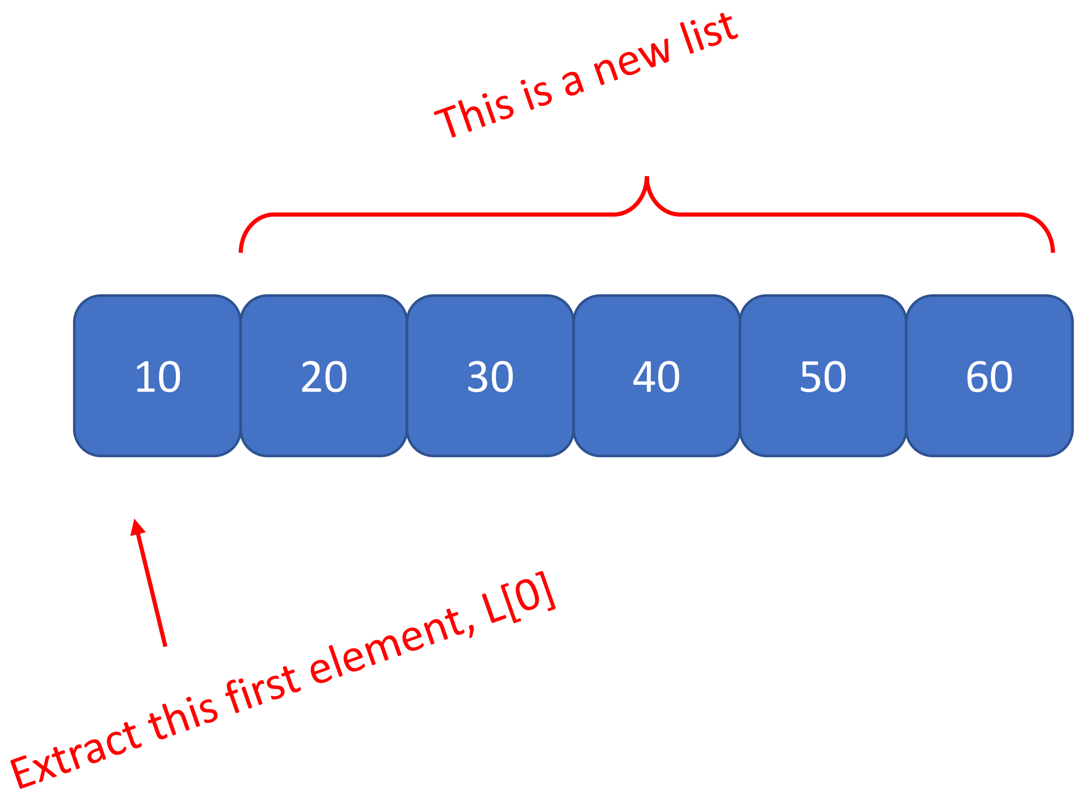
```
def total_iter(L):
  result = 0
  for e in L:
    result += e
  return result

test = [30, 40, 50]
print(total_iter(test))
```

VISUALIZING LISTS as RECURSIVE



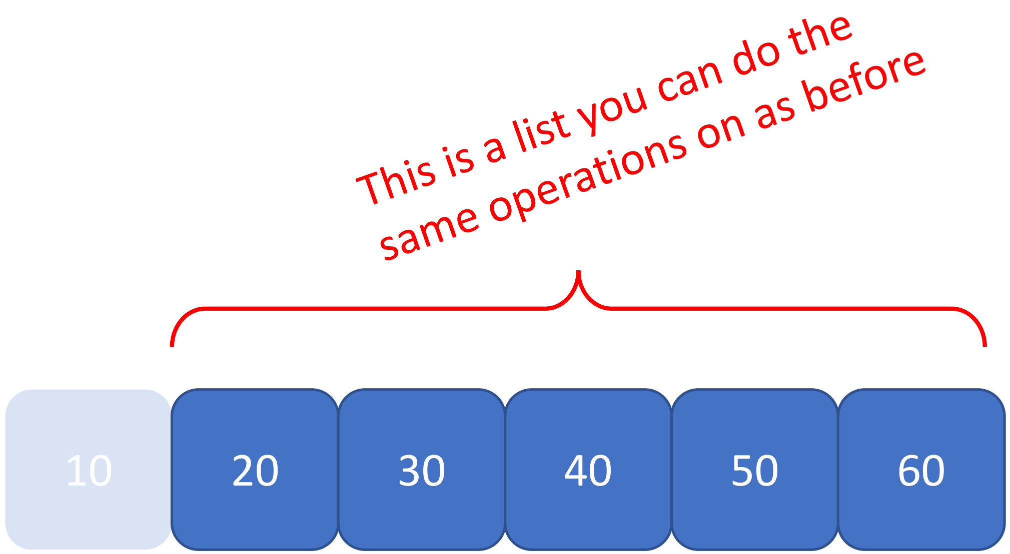
- Find sum of this original list

VISUALIZING LISTS as RECURSIVE



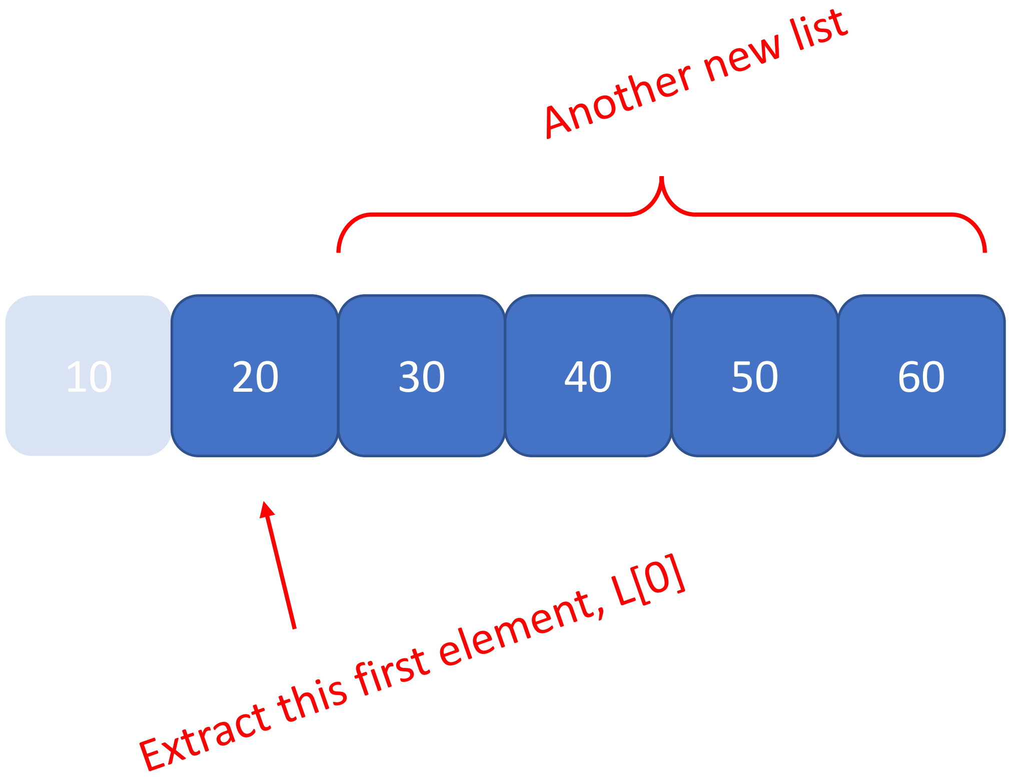
- L[0] + sum of the new list

VISUALIZING LISTS as RECURSIVE



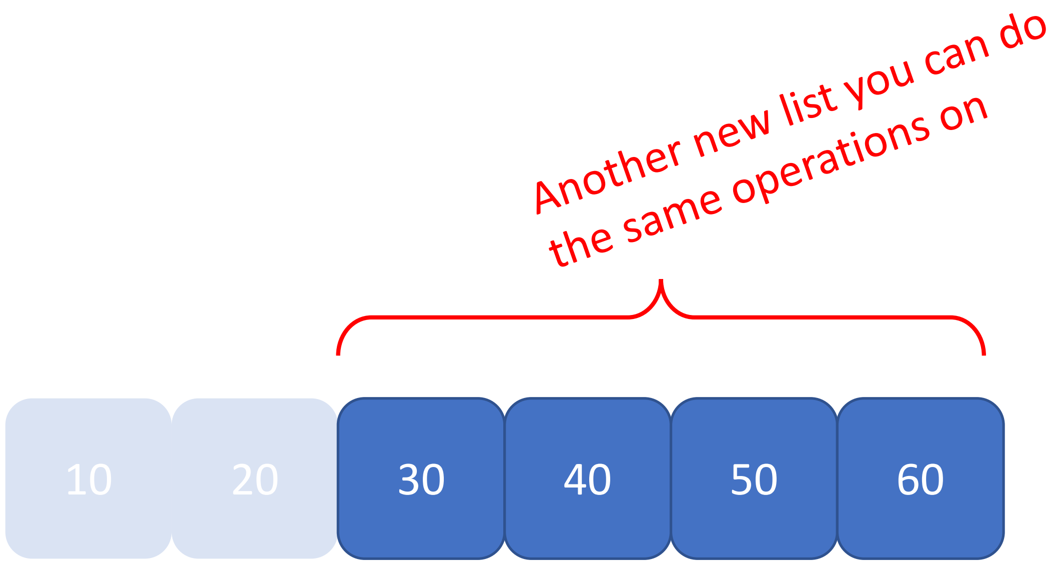
- Solve the same problem, slightly changed (its length is smaller)

VISUALIZING LISTS as RECURSIVE



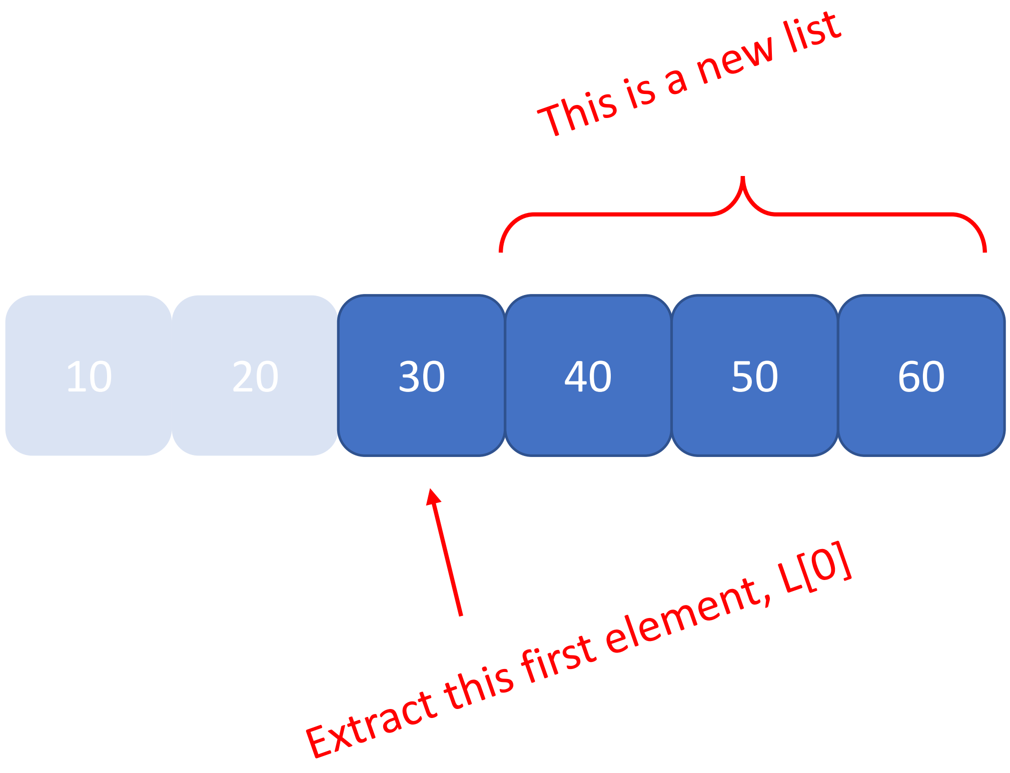
- L[0] + sum of the new list

VISUALIZING LISTS as RECURSIVE



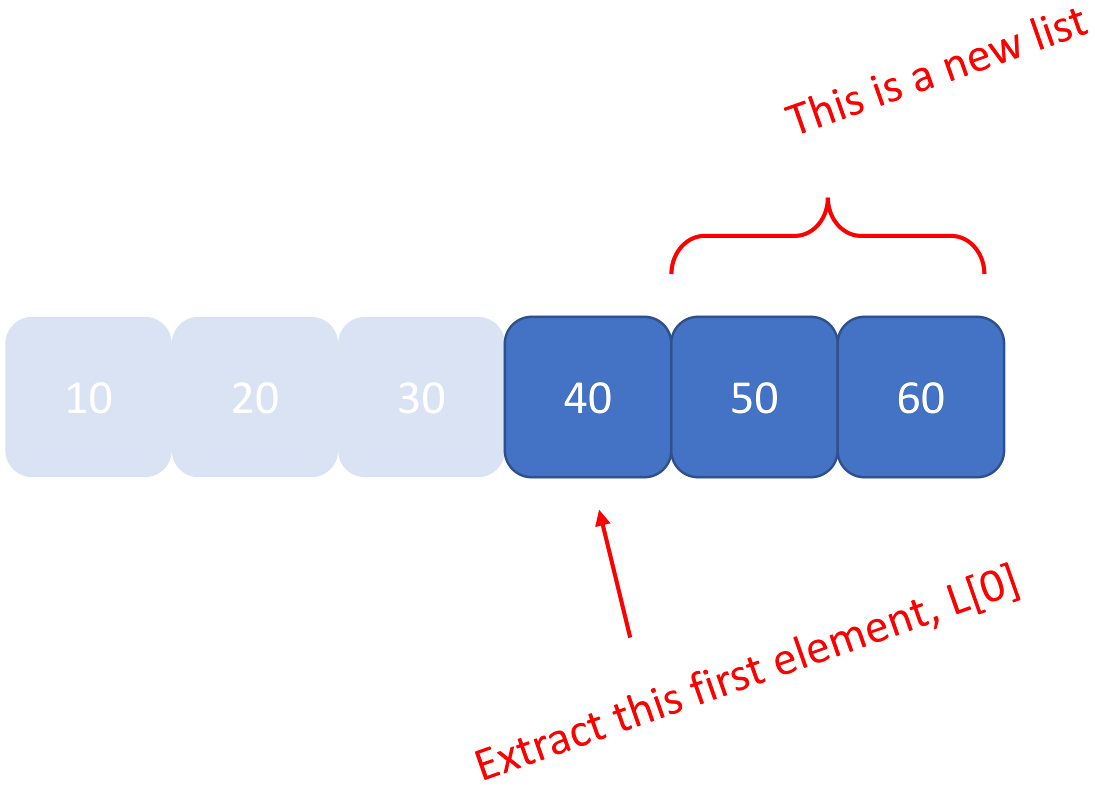
- Solve the same problem again, slightly changed

VISUALIZING LISTS as RECURSIVE



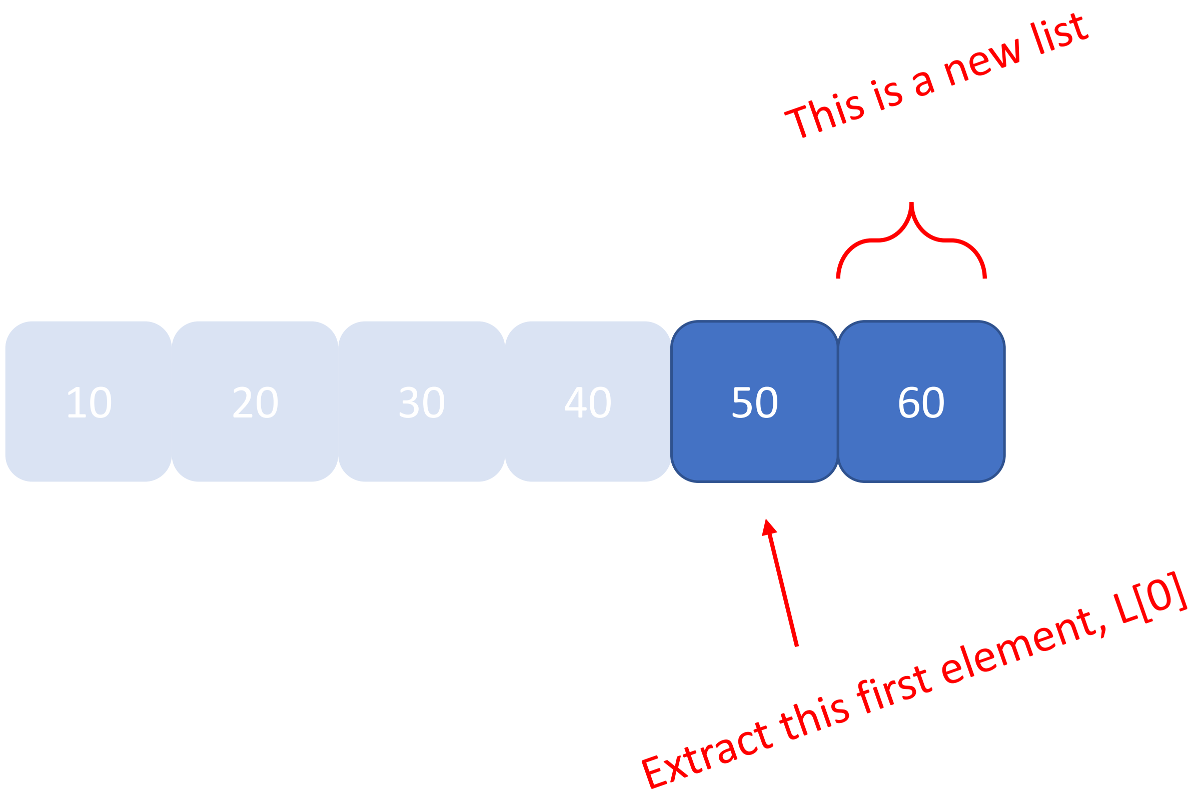
- L[0] + sum of the new list

VISUALIZING LISTS as RECURSIVE



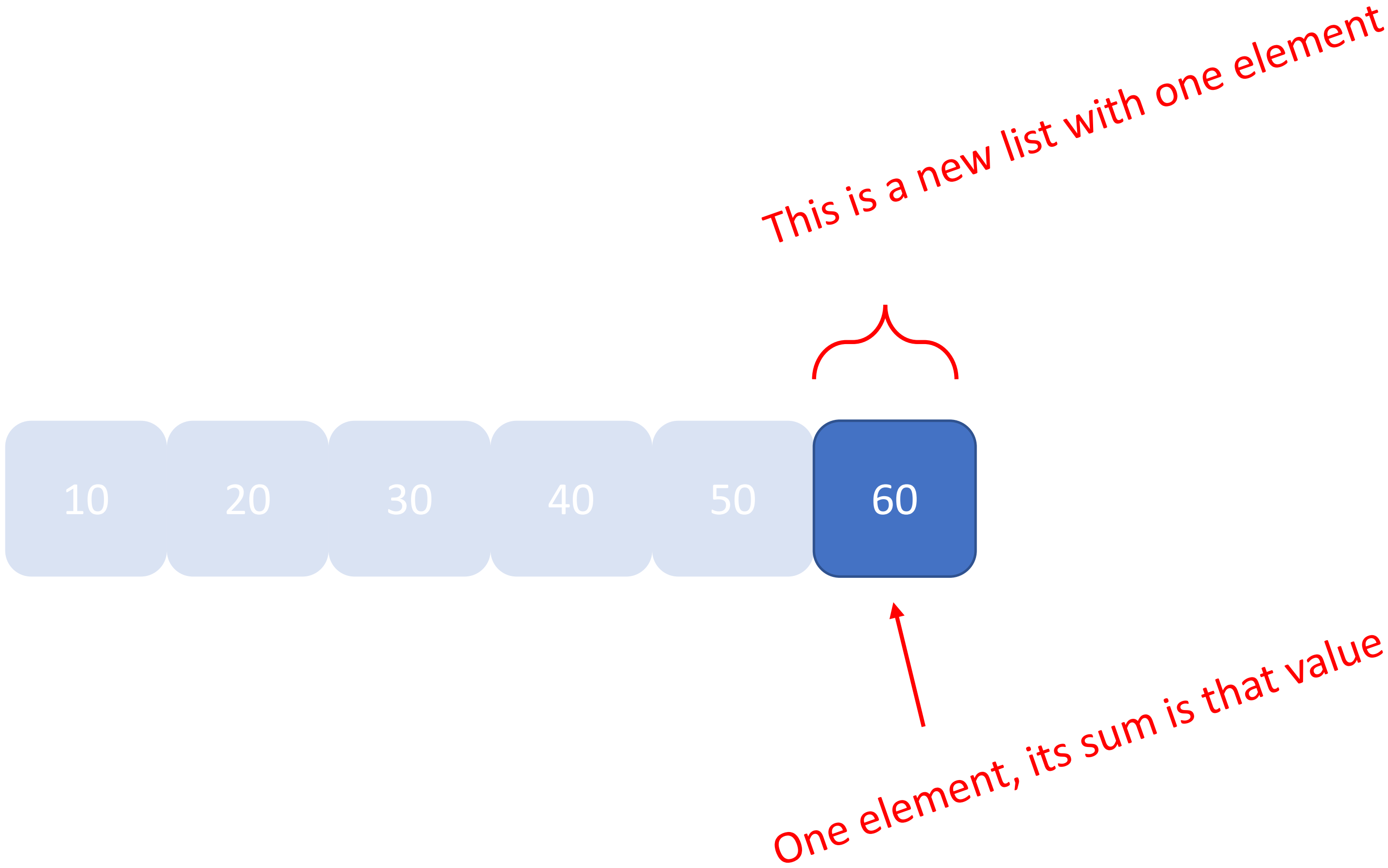
- Keep repeating, decreasing until a base case

VISUALIZING LISTS as RECURSIVE



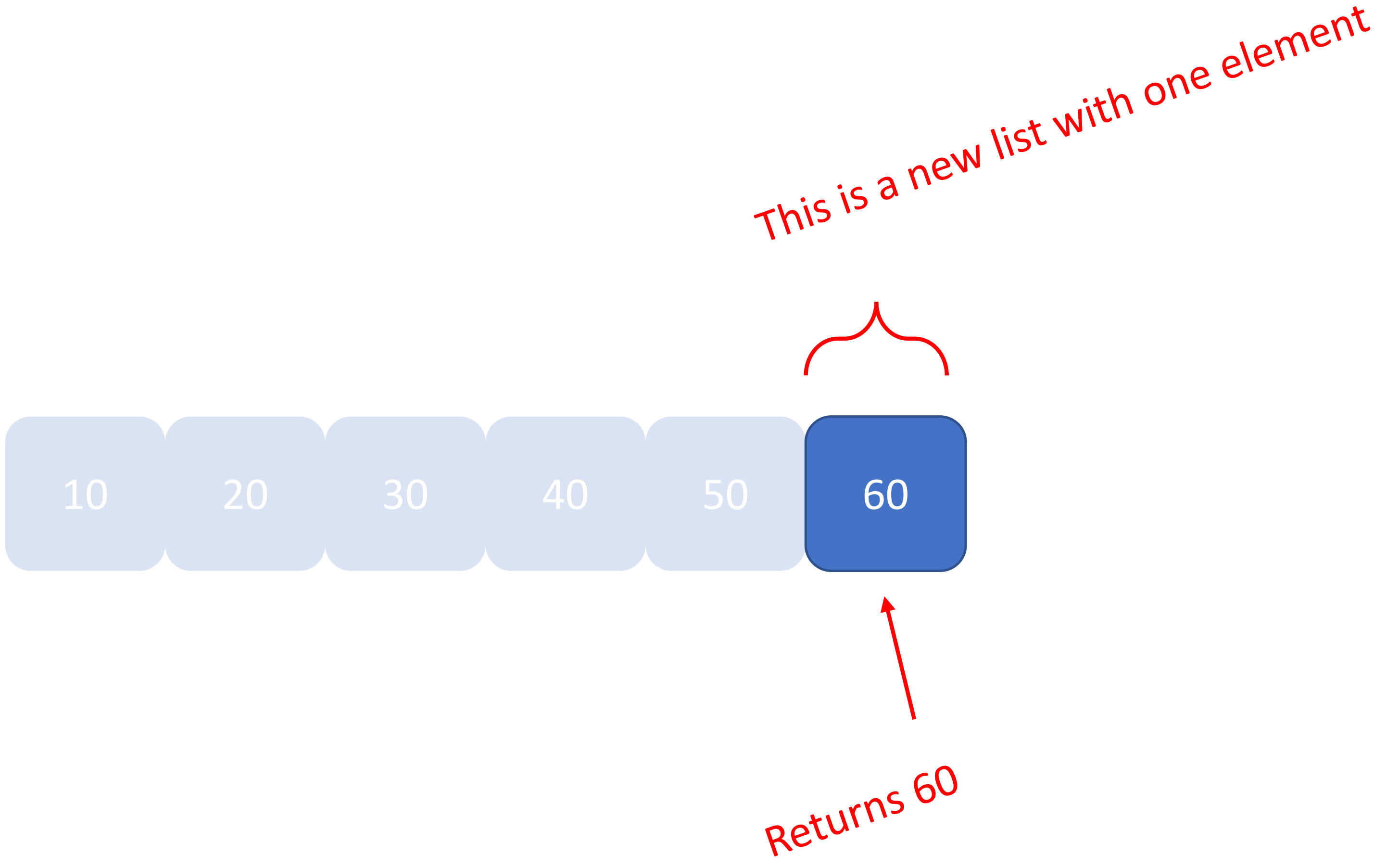
- Keep repeating, decreasing until a base case

VISUALIZING LISTS as RECURSIVE



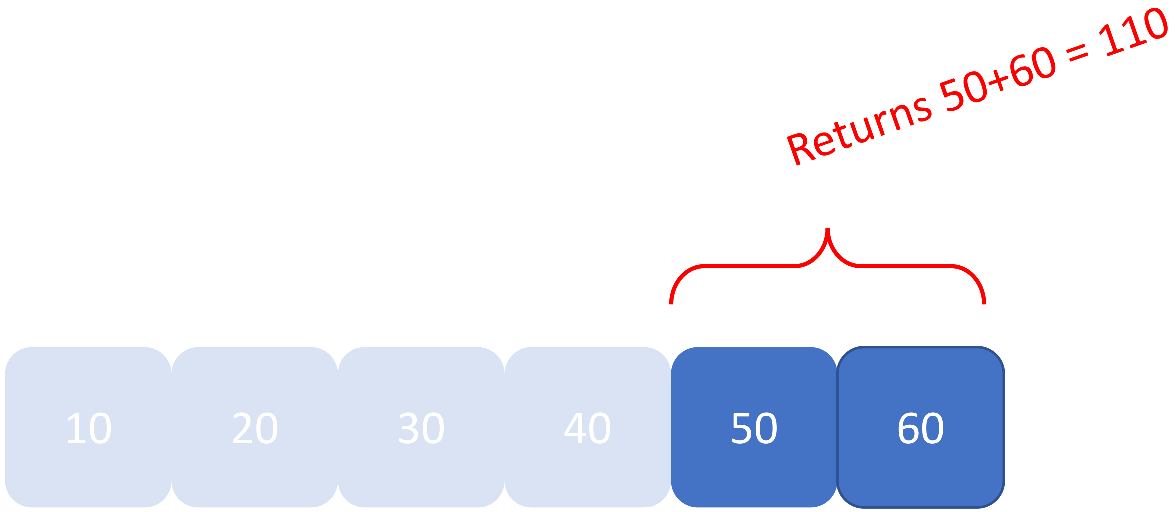
- The base case

VISUALIZING LISTS as RECURSIVE



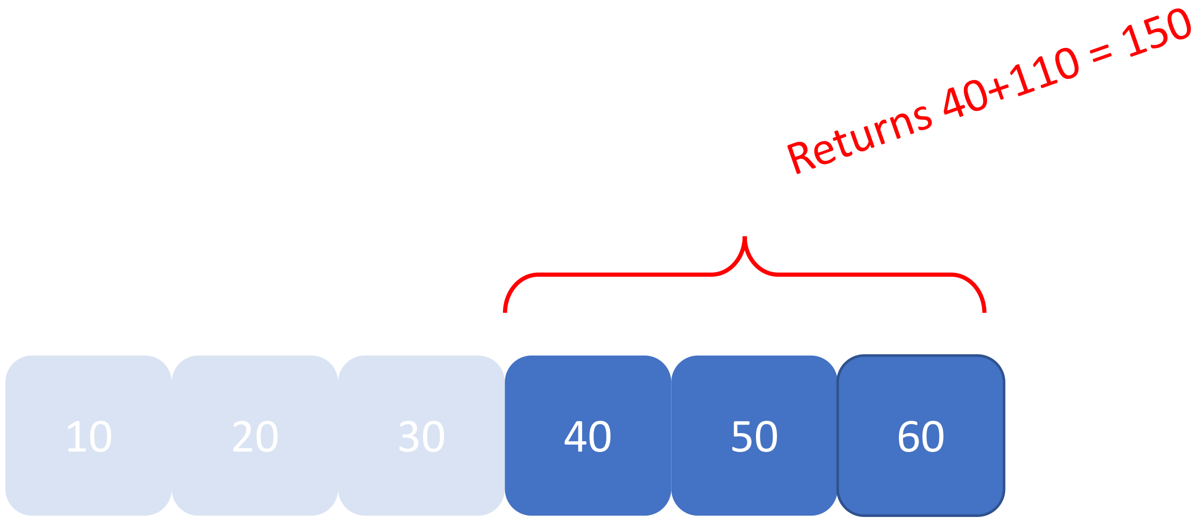
- Pass the sum back up the chain

VISUALIZING LISTS as RECURSIVE



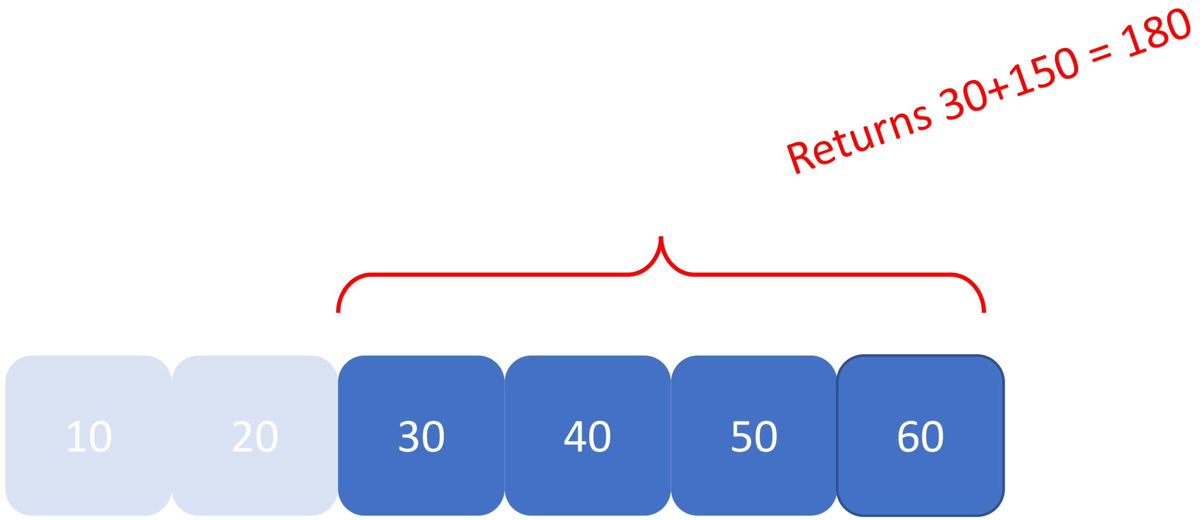
- Pass the sum back up the chain

VISUALIZING LISTS as RECURSIVE



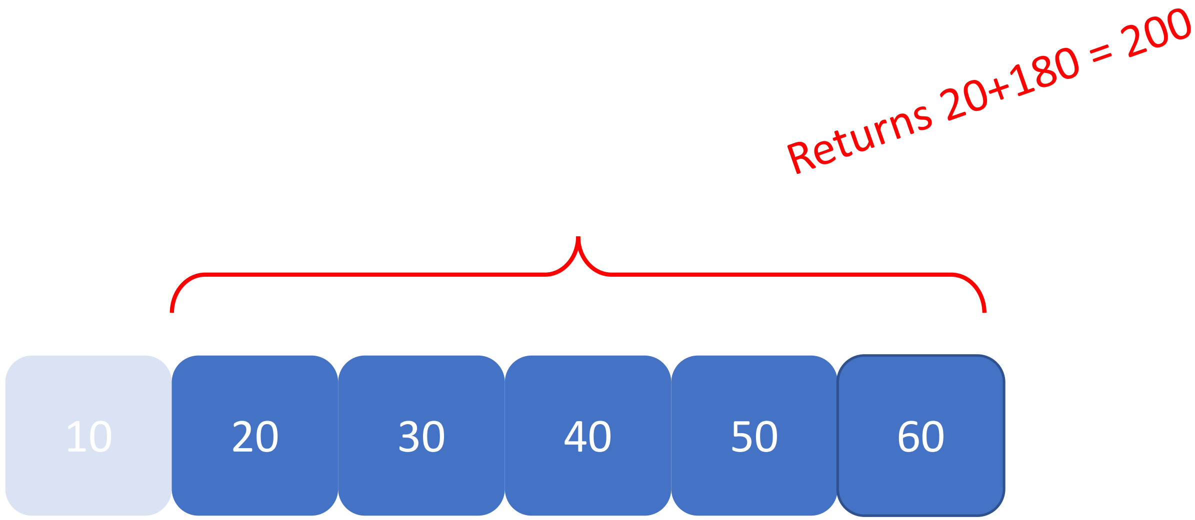
- Pass the sum back up the chain

VISUALIZING LISTS as RECURSIVE



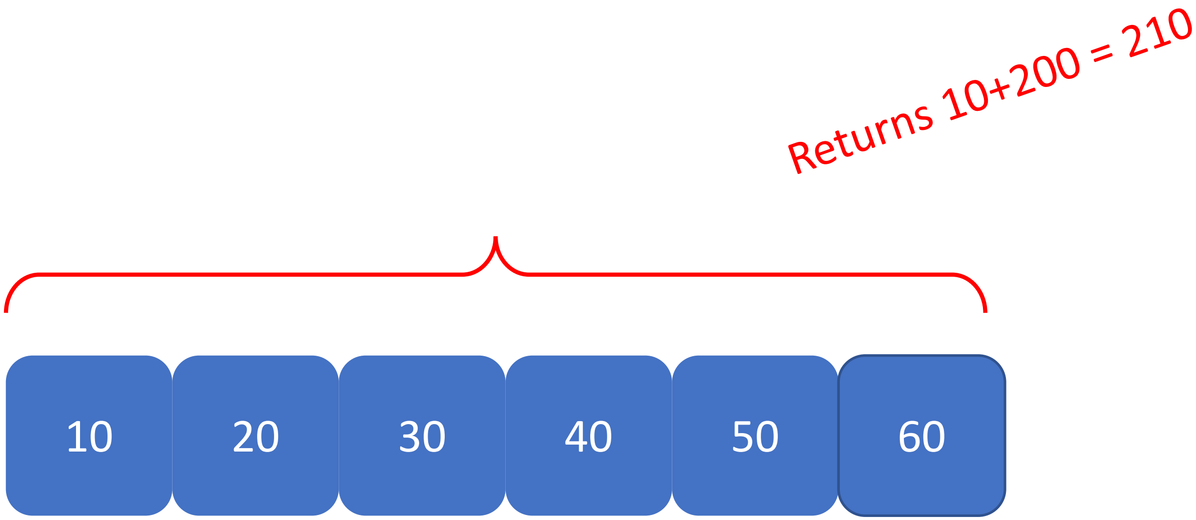
- Pass the sum back up the chain

VISUALIZING LISTS as RECURSIVE



- Pass the sum back up the chain

VISUALIZING LISTS as RECURSIVE



- Pass the sum back up the chain

SUM of LIST ELEMENTS: the PIECES

```
def total_recur(L):
    if

    else:



test = [30, 40, 50]
print(total_recur(test))
```

- Base case
- Recursive step

SUM of LIST ELEMENTS: the BASE CASE (one option)

```
def total_recur(L):
    if L == []:
        return 0
    else:


test = [30, 40, 50]
print(total_recur(test))
```

- What is the base case?
- One option:
  An **empty list** has sum 0

SUM of LIST ELEMENTS: the BASE CASE (another option)

```
def total_recur(L):
  if len(L) == 1:
    return L[0]
  else:



test = [30, 40, 50]
print(total_recur(test))
```

- What is the base case?
- Another option:
  A **list with one element** has a sum of that one element
- For example:
  `L = [50]`
  Returns:
  `50`

SUM of LIST ELEMENTS: the RECURSIVE STEP

```
def total_recur(L):
  if len(L) == 1:
    return L[0]
  else:
    return L[0] + # something

test = [30, 40, 50]
print(total_recur(test))
```

- What is the recursive step?
- Need to get to the base case somehow
- Let's look at elements one at a time
- Extract the **first one** and grab its value
- For example:
  `L = [30,40,50]`
  Returns:
  `30 + <something>`

SUM of LIST ELEMENTS RECURSIVE STEP will EVENTUALLY END

```
def total_recur(L):
    if len(L) == 1:
        return L[0]
    else:
        return L[0] + total_recur(L[1:])

test = [30, 40, 50]
print(total_recur(test))
```

- What is the recursive step?
- The function call finds the **sum of the remaining list elements**
- For example:
  `L = [30,40,50]`
  Returns:
  `30 + total_recur([40,50])`

SUM of LIST ELEMENTS: TAKEAWAYS, Python Tutor LINK

```
def total_recur(L):
  if len(L) == 1:
    return L[0]
  else:
    return L[0] + total_recur(L[1:])

test = [30, 40, 50]
print(total_recur(test))
```

- Notice:
- Every case in the function **returns something that is the same type**
  - Base case returns an int
  - Recursive step returns an int
- We need to **trust** that the recursive calls eventually do the right thing

YOU TRY IT!

- Modify the code we wrote to return the total length of all strings inside L:

```
def total_len_recur(L):
    if len(L) == 1:
        return _______
    else:
        return ___________________

test = ["ab", "c", "defgh"]
print(total_recur(test))  # prints 8
```

LOOKING for an ELEMENT in a LIST

ANOTHER EXAMPLE: Is an ELEMENT in a LIST? (careful with this implementation)

```
def in_list(L, e):
  if len(L) == 1:
    return L[0] == e
  else:
    return in_list(L[1:], e)
```

- Let's start by following the **same pattern** as the prev example
- Base case is when we have **one element**
  - Check if it's the one we are looking for
- Recursive step looks at the **remaining elements**
  - Grab the list from index 1 onward and look for e in it

ANOTHER EXAMPLE: Is an ELEMENT in a LIST? (careful with this implementation) Python Tutor

```
def in_list(L, e):
  if len(L) == 1:
    return L[0] == e
  else:
    return in_list(L[1:], e)

test = [2,5,8,1]
print(in_list(test, 1))

test = [2,1,5,8]
print(in_list(test, 1))
```

- Test it out
- test = [2,5,8,1] and e=1 gives **True**
  - **ok**
- test = [2,1,5,8] and e=1 gives **False**
  - **Not ok!**
- It checks only if the last elem is the one we are looking for!

ANOTHER EXAMPLE: Is an ELEMENT in a LIST? (fix the implementation)

```
def in_list(L, e):
  if len(L) == 1:
    return L[0] == e
  else:
    # Check the first element
    # before looking in the rest


      return in_list(L[1:], e)
```

- Still want to look at elements one at a time
- Need to check whether the element we extracted is the one we are looking for **at each function call**

ANOTHER EXAMPLE: Is an ELEMENT in a LIST? (fix the implementation)

```
def in_list(L, e):
  if len(L) == 1:
    return L[0] == e
  else:
    if L[0] == e:
      return True
    else:
      return in_list(L[1:], e)
```

- Still want to look at elements one at a time
- Add the **check in the recursive step**, before checking the rest of the list.

ANOTHER EXAMPLE:
Is an ELEMENT in a LIST?
(test the implementation) Python Tutor LINK

```
def in_list(L, e):
  if len(L) == 1:
    return L[0] == e
  else:
    if L[0] == e:
      return True
    else:
      return in_list(L[1:], e)
```

- Test it now
- test = [2,5,8,1] and e=1 gives True
  - **ok**
- test = [2,1,5,8] and e=1 gives True
  - **ok**
- test = [2,5,8] and e=1 gives False
  - **ok**

ANOTHER EXAMPLE:
Is an ELEMENT in a LIST?
(improve the implementation)

```
def in_list(L, e):
  if len(L) == 0:
    return False
  elif L[0] == e:
    return True
  else:
    return in_list(L[1:], e)
```

- Two cases that return L[0]
- Add case when L is empty
- **Simplify the code** to check the first element as another base case

BIG IDEA

Each case (base cases, recursive step) must return the same type of object.

Remember that function returns build upon each other!

If the base case returns a bool and the recursive step returns an int, this gives a type mismatch error at runtime.

FLATTEN a LIST with ONLY ONE LEVEL of LIST ELEMENTS

FLATTEN a LIST CONTAINING LISTS of ints e.g. [[1, 2],[3, 4],[9, 8, 7]] gives [1, 2, 3, 4, 9, 8, 7]

```
def flatten(L):
    if len(L) == 1:

    else:
```

- Base case
- There is only **one element** in L
- For example: [[2,3,4]]

FLATTEN a LIST CONTAINING LISTS of ints
e.g. `[[1, 2],[3, 4],[9, 8, 7]]`
gives `[1, 2, 3, 4, 9, 8, 7]`

```
def flatten(L):
    if len(L) == 1:
        return L[0]
    else:
```

- Base case
- Return that element
- For example:
  `[[2,3,4]]`
  Returns:
  `[2,3,4]`

FLATTEN a LIST CONTAINING LISTS of ints

e.g. `[[1, 2],[3, 4],[9, 8, 7]]`
gives `[1, 2, 3, 4, 9, 8, 7]`

```
def flatten(L):
  if len(L) == 1:
    return L[0]
  else:
    return L[0] + #something
```

- Recursive step
- Recall that + between two lists concatenates the elements into a new list
- Make a **new list containing the first element** and…

FLATTEN a LIST CONTAINING LISTS of ints

e.g. `[[1, 2],[3, 4],[9, 8, 7]]`
gives `[1, 2, 3, 4, 9, 8, 7]`
Python Tutor LINK

```
def flatten(L):
  if len(L) == 1:
    return L[0]
  else:
    return L[0] + flatten(L[1:])
```

- Recursive step
- ... **flatten the rest** of the remaining list
- For example:
  `[[1,2],[3,4],[9,8,7]]`
  Returns:
  `[1,2] + flatten([[3,4], [9,8,7]])`

YOU TRY IT!

- Write a recursive function according to the specs below.

```
def in_list_of_lists(L, e):
    """
    L is a list whose elements are lists containing ints.
    Returns True if e is an element within the lists of L
    and False otherwise.
    """
    # your code here

test = [[1,2], [3,4], [5,6,7]]
print(in_list_of_lists(test, 0))  # prints False
test = [[1,2], [3,4], [5,6,7]]
print(in_list_of_lists(test, 3))  # prints True
```

WHEN to USE RECURSION

- So far you should have some intuition for how to write recursive functions
- The problem is that so far you've been writing recursive version of functions that are usually easier to implement WITHOUT recursion :(
- So why learn recursion?
  - Some problems are very difficult to solve with iteration

INTUITION for WHEN to use RECURSION

- Remember when we learned while loops?
- Remember when we tried to write a program that kept asking the user which way to go in the Lost Woods of Zelda?
- We did not know ahead of time how many times we needed to loop! (aka **how many levels of if/else we needed**)
- While loops kept iterating as long as some condition held true.

```
if <exit right>:
    <set background to woods_background>
    if <exit right>:
        <set background to woods_background>
        if <exit right>:
            <set background to woods_background>
            and so on and on and on...
        else:
            <set background to exit_background>
    else:
        <set background to exit_background>
else:
    <set background to exit_background>
```


INTUITION for WHEN to use RECURSION

- In the list recursion examples so far, we knew how many levels we needed to iterate.
  - Either look at elems directly or in one level down
- **But lists can have elements that are lists, which can in turn have elements that are lists, which can in turn have elements that are lists, etc.**
- How can we use iteration to do these checks? It's hard.

```
for i in L:
    if type(i) == list:
        for j in i:
            if type(j) == list:
                for k in j:
                    if type(k) == list:
                        # and so on and on
                    else:
                        # do what you need to do
            else:
                # do what you need to do
    else:
        # do what you need to do
# done with the loop over L and all its elements
```

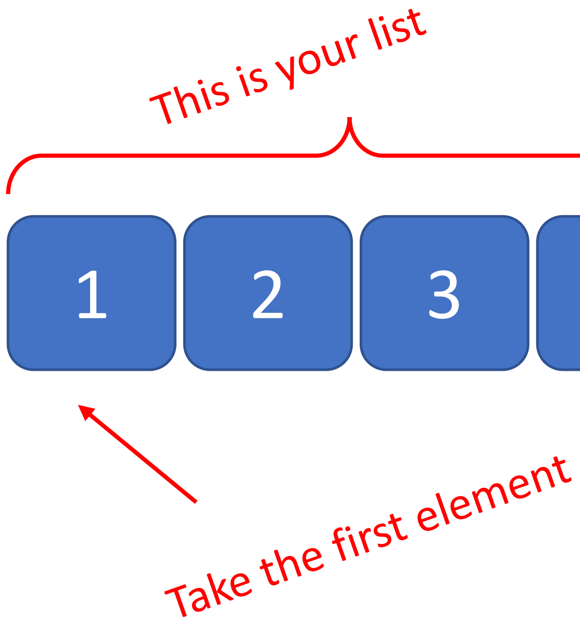
You don't know how deep this goes

PROBLEMS that are NATURALLY RECURSIVE

- A file system
- Order of operations in a calculator
- Scooby Doo gang searching a haunted castle
- Bureaucracy

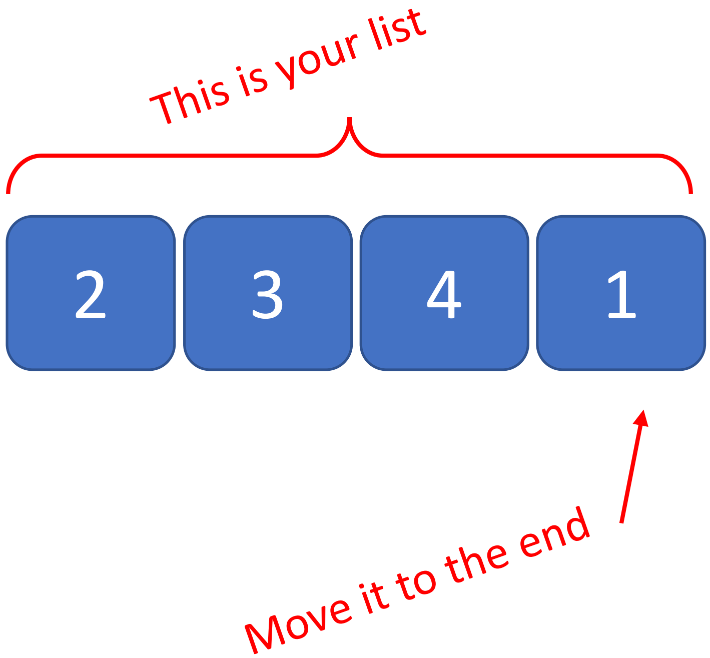
LET'S SEE HOW TO GO FROM ONE LEVEL to MANY LEVELS (RECURSIVELY)

- Example: reverse a list's elements
- How to break up the problem into a smaller version of your same problem?



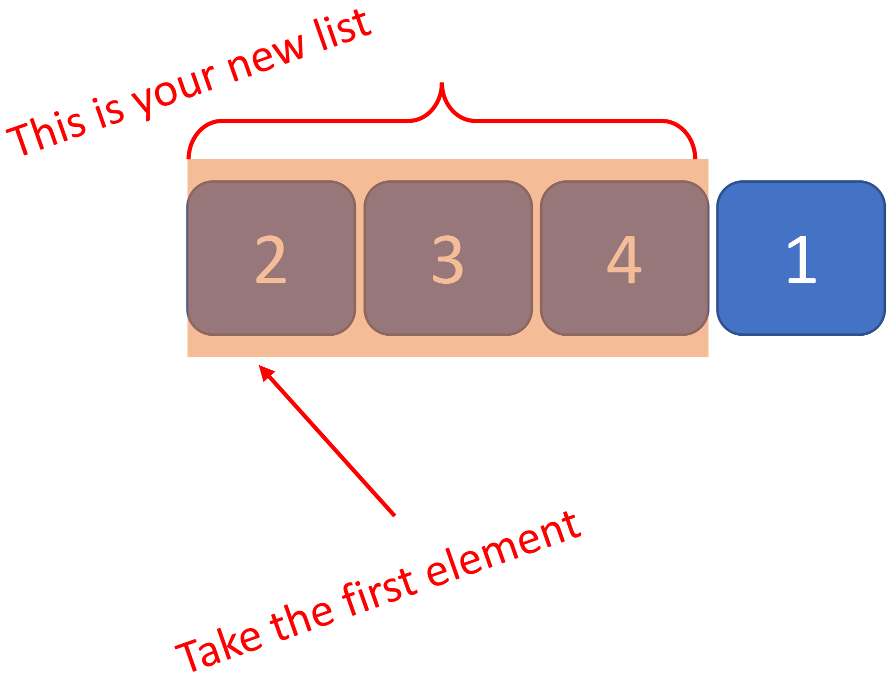
LET'S SEE HOW TO GO FROM ONE LEVEL to MANY LEVELS (RECURSIVELY)

- Example: reverse a list's elements
- How to break up the problem into a smaller version of your same problem?



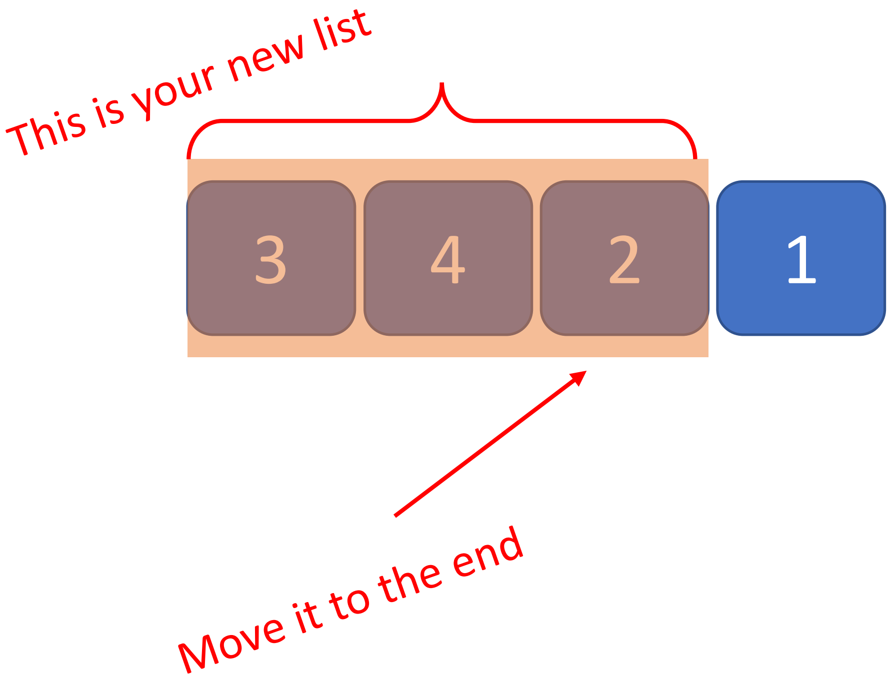
LET'S SEE HOW TO GO FROM ONE LEVEL to MANY LEVELS (RECURSIVELY)

- Example: reverse a list's elements
- How to break up the problem into a smaller version of your same problem?



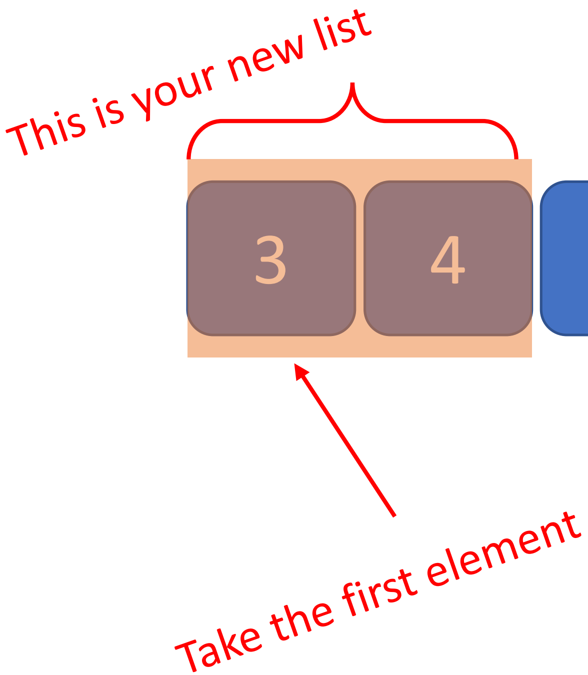
LET'S SEE HOW TO GO FROM ONE LEVEL to MANY LEVELS (RECURSIVELY)

- Example: reverse a list's elements
- How to break up the problem into a smaller version of your same problem?



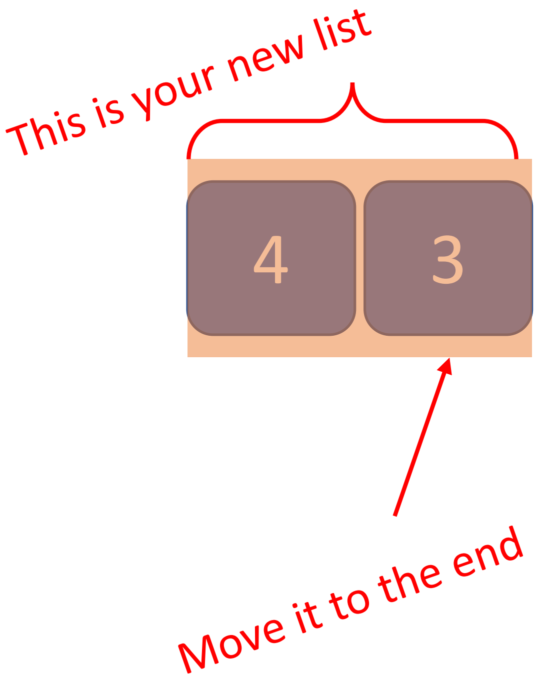
LET'S SEE HOW TO GO FROM ONE LEVEL to MANY LEVELS (RECURSIVELY)

- Example: reverse a list's elements
- How to break up the problem into a smaller version of your same problem?



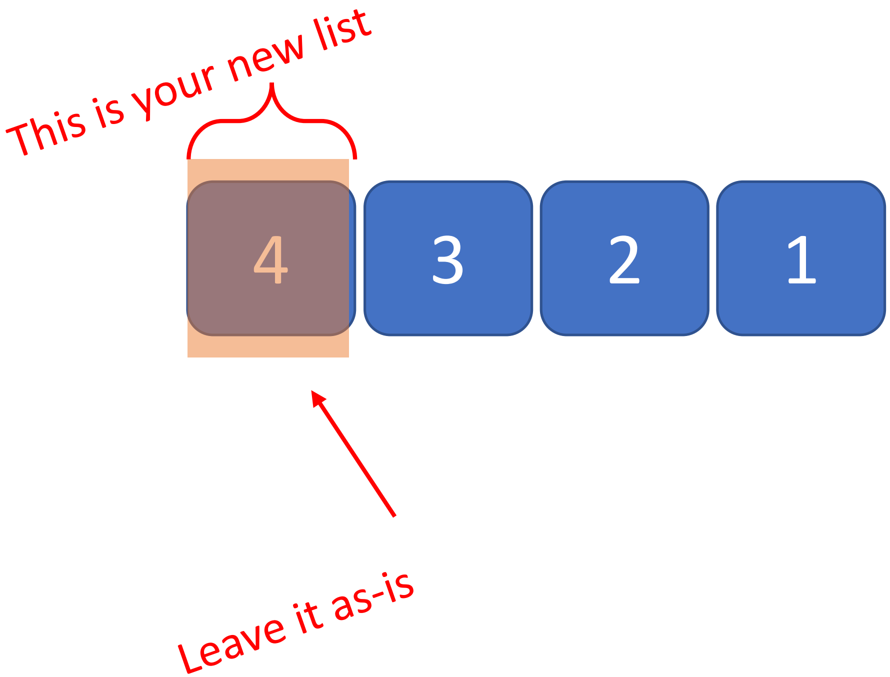
LET'S SEE HOW TO GO FROM ONE LEVEL to MANY LEVELS (RECURSIVELY)

- Example: reverse a list's elements
- How to break up the problem into a smaller version of your same problem?



LET'S SEE HOW TO GO FROM ONE LEVEL to MANY LEVELS (RECURSIVELY)

- Example: reverse a list's elements
- How to break up the problem into a smaller version of your same problem?



REVERSE a LIST of ELEMENTS: TOP-LEVEL ONLY

```
def my_rev(L):
    if len(L) == 1:

    else:
```

- Base case

REVERSE a LIST of ELEMENTS: TOP-LEVEL ONLY

```
def my_rev(L):
  if len(L) == 1:
    return L
  else:
```

- Base case
- Reversing a list with one element is just that list.

REVERSE a LIST of ELEMENTS: TOP-LEVEL ONLY

```
def my_rev(L):
  if len(L) == 1:
    return L
  else:
    return  <something>  + [L[0]]
```

Make a list with one element

- Recursive step
- Move element at index 0 to the end.
- Equivalent to **concatenating** something with that element
- For example:
  `[10,20,30,40]`
  Returns:
  `<something> + [10]`

REVERSE a LIST of ELEMENTS: TOP-LEVEL ONLY

```
def my_rev(L):
  if len(L) == 1:
    return L
  else:
    return my_rev(L[1:]) + [L[0]]
```

- Recursive step
- Solve the same problem, but on the **list containing all elements except the first** one
- For example:
  `[10,20,30,40]`
  Returns:
  `my_rev([20,30,40]) + [10]`

REVERSE a LIST of ELEMENTS: TOP-LEVEL ONLY

Python Tutor LINK

```
def my_rev(L):
  if len(L) == 1:
    return L
  else:
    return my_rev(L[1:]) + [L[0]]

test = [1, 2, "abc"]
print(my_rev(test))


test = [1,['d'],['e',['f', 'g']]]
print(my_rev(test))
```

- Test it

```
test = [1, 2, "abc"]
```

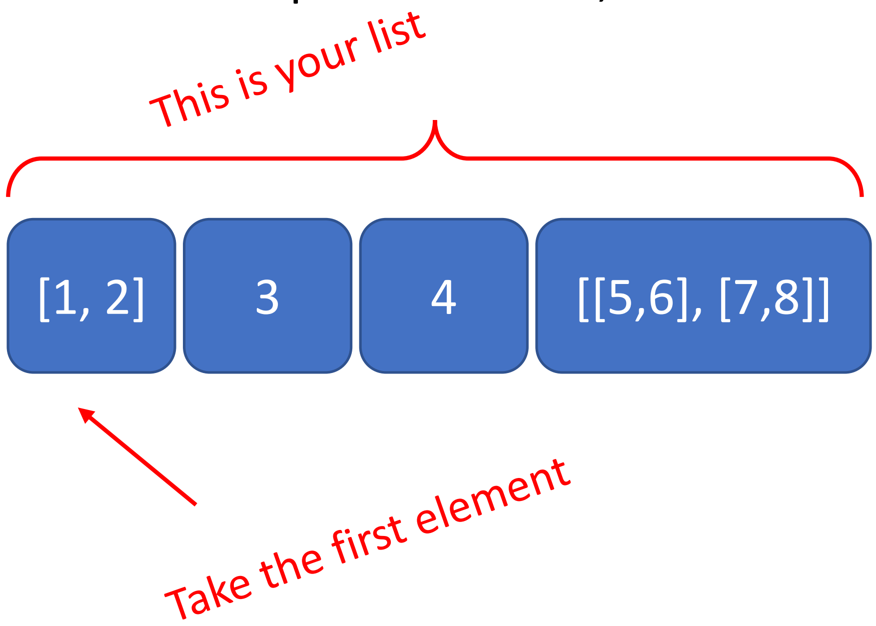
```
# prints
['abc', 2, 1]
```

```
test = [1,['d'],['e',['f', 'g']]]
```

```
# prints this, notice it
# just reverses top-level elems
[['e', ['f', 'g']], ['d'], 1]
```

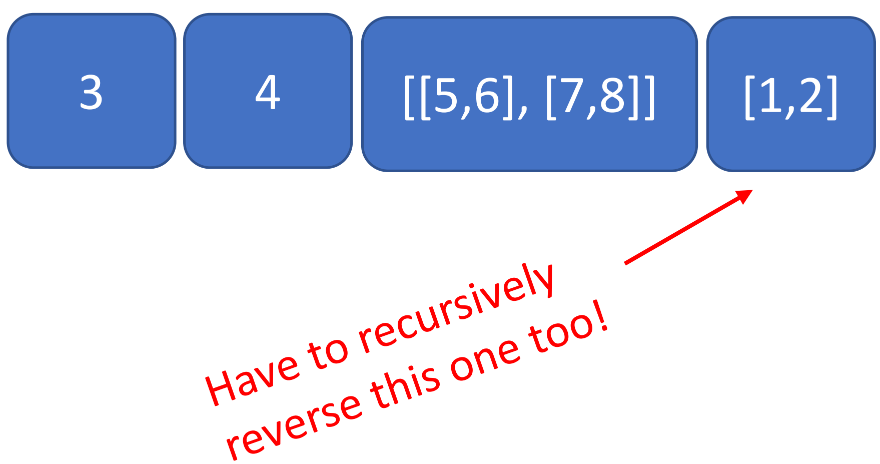
ALL ELEMENTS GET REVERSED

- Example: reverse all elements in all sublists
- Need to know whether we have an element or a list
  - Elements are put at the end, lists are reversed themselves



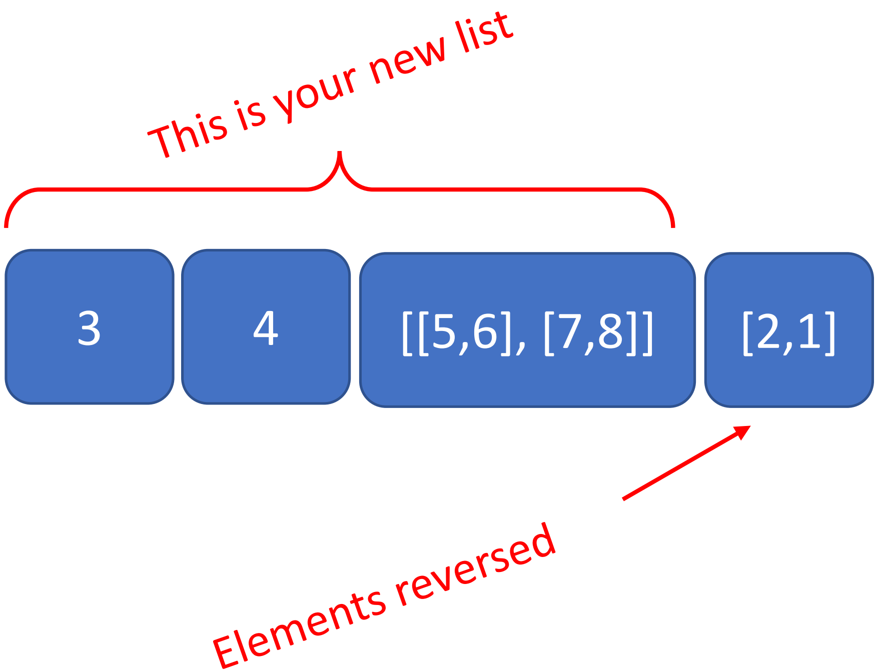
ALL ELEMENTS GET REVERSED

- If it's a list,



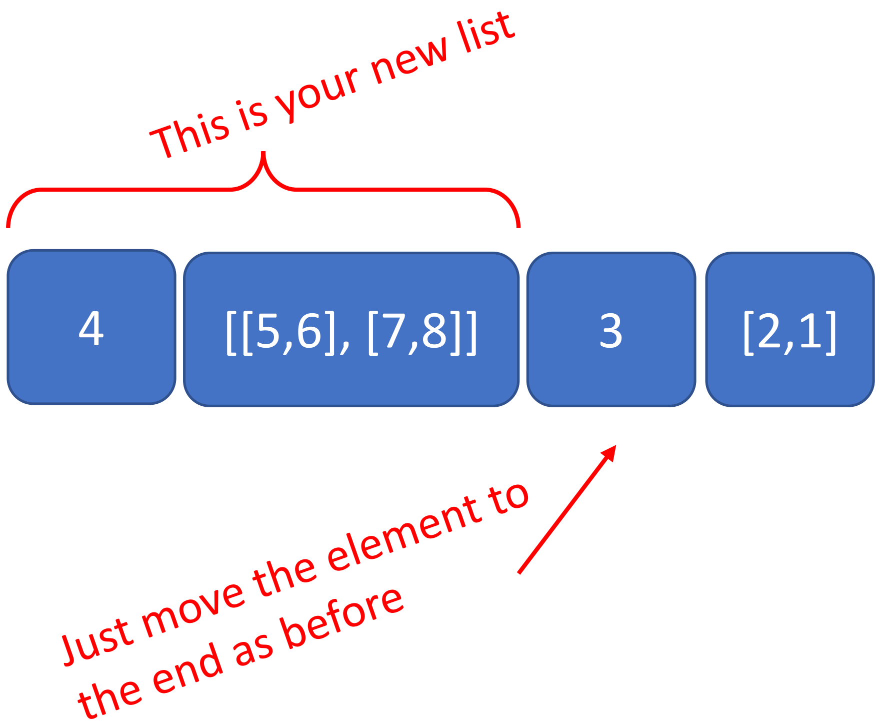
ALL ELEMENTS GET REVERSED

- If it's a list,



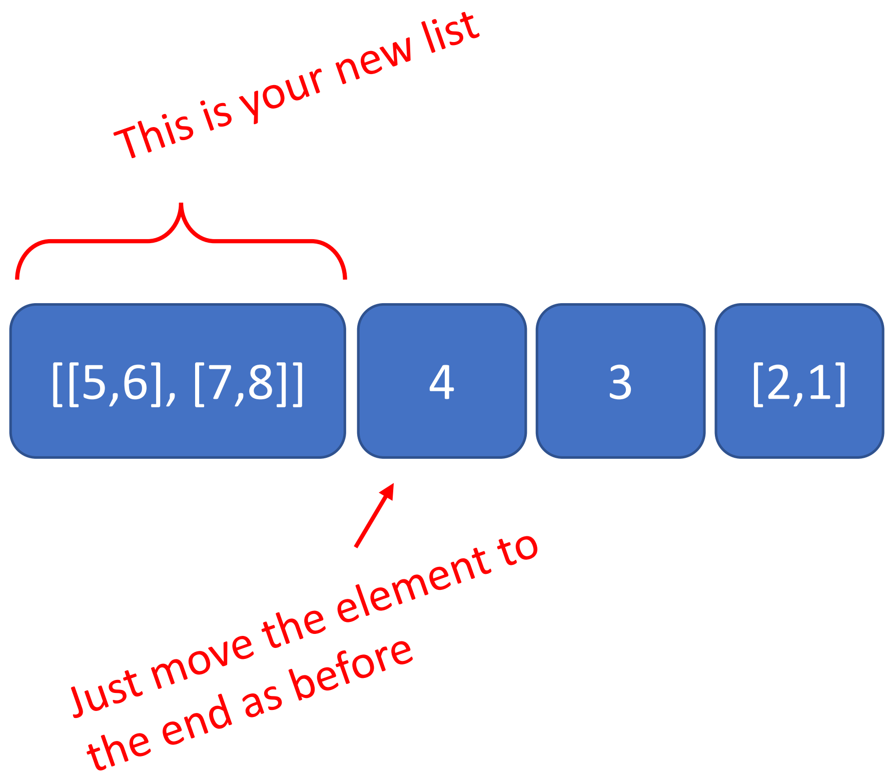
ALL ELEMENTS GET REVERSED

- If it's **not** a list



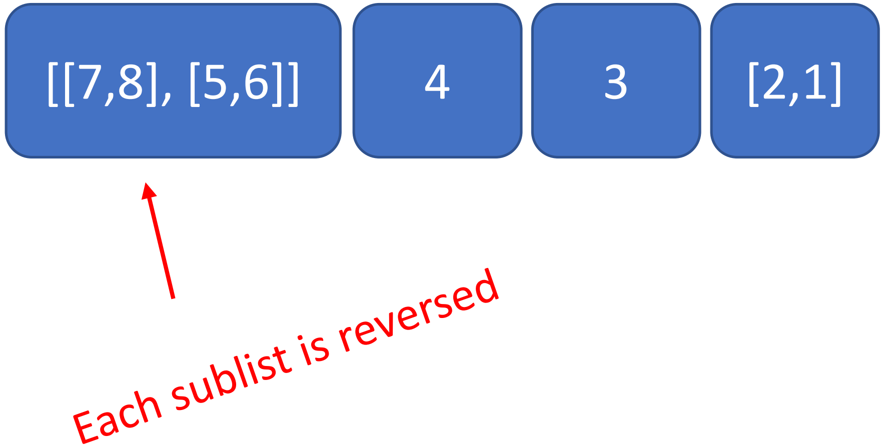
ALL ELEMENTS GET REVERSED

- And so on.



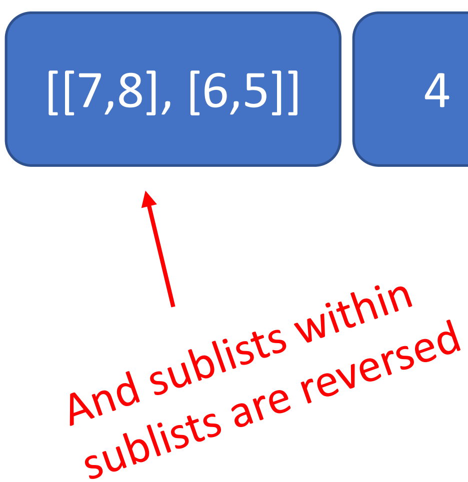
ALL ELEMENTS GET REVERSED

- Lists within lists get reversed each



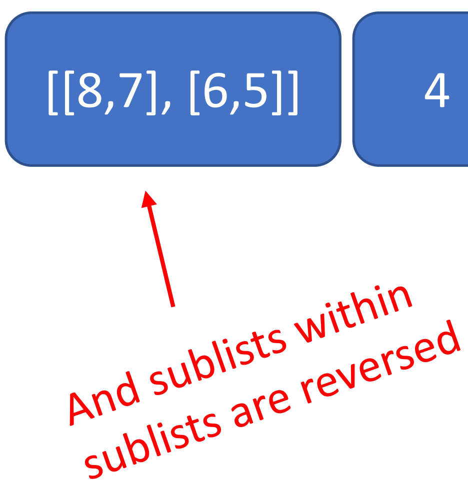
ALL ELEMENTS GET REVERSED

- Lists within lists get reversed each



ALL ELEMENTS GET REVERSED

- Lists within lists get reversed each



REVERSE a LIST of ELEMENTS: ALL ELEMENTS GET REVERSED

```
def deep_rev(L):
    if len(L) == 1:
        if type(L[0]) != list:
            # do something
        else:
            # do something
```

- Base case is NOT the same
- A single element can **either** be a
  - **Non-list**:
  - **List**:

REVERSE a LIST of ELEMENTS: ALL ELEMENTS GET REVERSED

```
def deep_rev(L):
    if len(L) == 1:
        if type(L[0]) != list:
            return L
        else:
            # do something
```

- Base case is NOT the same
- A single element can **either** be a
  - **Non-list**: it's just the list itself, like before
  - **List**:

REVERSE a LIST of ELEMENTS: ALL ELEMENTS GET REVERSED

```
def deep_rev(L):
  if len(L) == 1:
    if type(L[0]) != list:
      return L
    else:
      return [deep_rev(L[0])]
```

Make a list with one element

- Base case is NOT the same
- A single element can **either** be a
  - **Non-list**: it's just the list itself, like before
  - **List**: Must reverse it!

REVERSE a LIST of ELEMENTS: ALL ELEMENTS GET REVERSED

```
def deep_rev(L):
  if len(L) == 1:
    if type(L[0]) != list:
      return L
    else:
      return [deep_rev(L[0])]
  else:
      if type(L[0]) != list:
        # do something
      else:
        # do something
```

- **Recursive** step
- Extract the first element. It can **either** be a
  - **Non-list**:
  - **List**:

REVERSE a LIST of ELEMENTS: ALL ELEMENTS GET REVERSED

```
def deep_rev(L):
  if len(L) == 1:
    if type(L[0]) != list:
      return L
    else:
      return [deep_rev(L[0])]
  else:
      if type(L[0]) != list:
        return deep_rev(L[1:]) + [L[0]]
      else:
        # do something
```

Make a list with one element

- **Recursive** step
- Extract the first element. It can **either** be a
  - **Non-list**: reverse the remaining elements and concatenate the result with the first element
  - **List**:

REVERSE a LIST of ELEMENTS: ALL ELEMENTS GET REVERSED

```
def deep_rev(L):
    if len(L) == 1:
        if type(L[0]) != list:
            return L
        else:
            return [deep_rev(L[0])]
    else:
        if type(L[0]) != list:
            return deep_rev(L[1:]) + [L[0]]
        else:
            return deep_rev(L[1:]) + [deep_rev(L[0])]
```

- **Recursive** step
- Extract the first element. It can **either** be a
  - **Non-list**: reverse the remaining elements and concatenate the result with the first element
  - **List**: reverse the remaining elements and concatenate the result with the first element reversed (it's a list!) too

Make a list with one element

REVERSE a LIST of ELEMENTS: ALL ELEMENTS GET REVERSED
CLEANED UP CODE

```
def deep_rev(L):
  if L == []:
    return []
  elif type(L[0]) != list:
    return deep_rev(L[1:]) + [L[0]]
  else:
    return deep_rev(L[1:]) + [deep_rev(L[0])]
```

- Extract out the **empty list**
- Extract out **L[0]**

BIG IDEA

Recursion procedure from this lecture can be applied to any indexable ordered sequence.

The same idea will work on problems involving strings.

The same idea will work on problems involving tuples.

MAJOR RECURSION TAKEAWAYS

- Most problems are solved more **intuitively with iteration**
  - We show recursion on these to:
    - Show you a **different way of thinking** about the same problem (algorithm)
    - Show you **how to write a recursive function** (programming)
- Some problems have **nicer solutions with recursion**
  - If you recognize solving the same problem repeatedly, use recursion
- Tips
  - Every case in your recursive function **must return the same type of thing**
    i.e. don't have a base case `return []`
    and a recursive step `return len(L[0])+recur(L[1:])`
  - Your function **doesn't have to be efficient on the first pass**
    - It's ok to have more than 1 base case
    - It's **ok to break down the problem** into many if/elifs
    - As long as you are **making progress** towards a base case recursively

YOU TRY IT!

- I added many practice recursion questions in the .py file associated with this lecture, to prep for the quiz.
- 1) An exercise to implement a recursive function (**no** lists within lists etc.)
- 2) An exercise to implement a recursive function (**with** lists within lists within lists etc.)
- 3) Three buggy recursion implementations to fix.