CSc 120
Introduction to Computer Programming II

06: Recursion

Problem

How much money is in this cup?

Approach:

- We will consider a *different* way of adding up the coins
- The TAs will demonstrate!

How much money is in this cup?

If the cup is not empty:

Take out a coin. Pass the cup to the next person and ask them:

"How much money is in this cup?"

When they answer, add your coin to their answer and pass your answer back

your_answer = your_coin + their_answer

else the cup is empty:

Answer "zero" to the person who passed to you.

your_answer = 0

Challenge

Can we express that algorithm/process in Python?

Idea:

```
>>> cup = [5, 10, 1, 5]
>>> how_much_money(cup)
21
```

Write Python code that models the cup passing example.

function: how_much_money

```
def how_much_money(cup):
    if cup == []:
        return 0
    else:
```

function: how_much_money

```
def how_much_money(cup):
    if cup == []:
        return 0
    else:
        return cup[0] + how_much_money(cup[1:])
```

5 [10, 1, 5]

Usage:

```
>>> how_much_money([5, 10, 1, 5])
21
```

Calls and returns

```python
def how_much_money(cup):
    if cup == []:
        return 0
    else:
        return cup[0] + how_much_money(cup[1:])
```

```
how_much_money([5,10,1,5])
| how_much_money([10,1,5])
| | how_much_money([1,5])
| | | how_much_money([5])
| | | | how_much_money([])
| | | | how_much_money returned 0
| | | how_much_money returned 5
| | how_much_money returned 6
| how_much_money returned 16
how_much_money returned 21
```

Manual expansion of calls

```
>>> 5 + how_much_money([10, 1, 5])
21
>>> 5 + (10 + how_much_money([1,5]))
21
>>> 5 + (10 + (1 + how_much_money([5])))
21
>>> 5 + (10 + (1 + (5 + how_much_money([]))))
21
```

Recursion

A function is *recursive* if it calls itself:

```
def  how_much_money( ... ):
        ...
        how_much_money( ... )  ← recursive call
        ...
```

The call to itself is a *recursive call*

Recursion

A solution to a problem is *recursive* when it is constructed from the solution to a simpler version of the same problem.

Recursion

A solution to a problem is *recursive* when it is constructed from the solution to a simpler version of the same problem.

```
def how_much_money(cup):
    if cup == []:
        return 0
    else:
        return cup[0] + how_much_money(cup[1:])
```

simpler version of the problem (or reduced data)

Recursion

- Recursive functions have two kinds of cases:
  - *base case(s)* :
    - *do some trivial computation and return the result*
  - *recursive case(s)* :
    - *the expression of the problem is a simpler case of the same problem*
    - *the input is reduced or the size of the problem is reduced*
- *Note*: the recursive call is given a smaller problem to work on
  - e.g., it makes progress towards the base case

recursion: base case/recursive case

```
def how_much_money(cup):
    if cup == []:
        return 0
    else:
        return cup[0] + how_much_money(cup[1:])
```

base case:
cup == []

recursive
case:
cup != []

The convention is to handle the base case(s) first.

Problem 1

Write a recursive function to count the number of coins in a cup. *The len function is not allowed.*

Usage:

```
>>> count_coins([10, 5, 1, 5])
4
```

Solution

```
def count_coins(cup):
    if cup == []:
        return 0
    else
        return 1 + count_coins(cup[1:])
```

Solution

base case:
cup == []

```
def count_coins(cup):
    if cup == []:
        return 0
    else:
        return 1 + count_coins(cup[1:])
```

recursive case:
cup != []

recursive call is on a smaller problem

Problem 2

Write a recursive function to count the number of nickels in a cup.

Usage:

```
>>> count_nickels([10, 5, 1, 5, 1])
2
```

Solution

base case:
cup == []

recursive
case:
cup != []

```
def count_nickels(cup):
    if cup == []:
        return 0
    else:
        if cup[0] == 5:
            return 1 + count_nickels(cup[1:])
        else:
            return count_nickels(cup[1:])
```

recursive call is on a smaller problem

Problem 3

Write a recursive function that returns the total length of all the elements of a list of lists (a 2-d list).

Usage:

```
>>> total_length([[1,2], [8,2,3,4], [2,2,2]])
9
```

Solution

```
def total_length(alist):
    if alist == []:
        return 0
    else:
        return len(alist[0]) + total_length(alist[1:])
```

base case:
alist == []

recursive case:
alist != []

recursive call is on a smaller problem

Problem 4

Recall that factorial is defined by the equation:

$$n! = n * (n-1) * (n-2) * (n-3)* ... * 2 * 1$$

and

$$0! = 1$$

Write a recursive function that computes the factorial of a number.

Usage:

```
>>> fact(4)
24
```

Solution

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

base case:
n == 0

recursive case:
n != 0

recursive call is on a smaller problem

EXERCISE-ICA18- p. 3

Write a recursive function sum_list(alist) that returns the sum of the elements in alist.

Usage:

```
>>> sum_list([3, 5, 6, 1)
15
```

EXERCISE-ICA18- p. 4

Write a recursive function string_len(s) that returns the length of string s.

Usage:

```
>>> >>> string_len("I wandered lonely as a cloud")
28
>>>
```

Recursion how to: sumlist

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

base case

recursive case

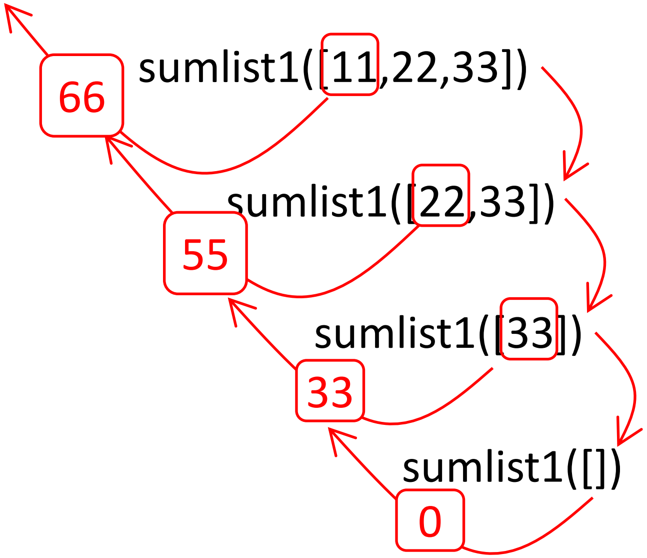
One round of computation adds L[0] and the result of the recursive call

argument to recursive call is "rest of the list" after L[0] (recurses on a smaller problem)

Recursion: flow of values

Version 1

```
def sumlist1(L):
    if len(L)  = 0:
        return 0
    else:
        return L[0] + sumlist1(L[1:])
```



EXERCISE-ICA19- p. 1

Write a recursive function join_all(alist) that takes a list alist and returns a string consisting of every element of alist concatenated together.

Usage:

```
>>> join_all([1,2,3,4,5])
'12345'
>>>
>>> join_all(['aa','bb'])
'aabb'
```

EXERCISE-ICA19-p. 2

Write a recursive function that implements join.

That is, write a function `join(alist, sep)` that takes a list `alist` and creates a string consisting of every element of `alist` separated by the string `sep`.

Usage:

```
>>> join(['aa', 'bb' , 'cc'], '-')
'aa-bb-cc'
```

the runtime stack

How recursion works

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


>>> fact(4)
24
```

How recursion works

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

>>> fact(4)
24
```

We need the value of n both before and after the recursive call

∴ its value has to be saved somewhere

"somewhere" ≡ "stack frame"

How recursion works

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


>>> fact(4)
24
```

Python's runtime system* maintains a stack:

- push a "frame" when a function is called
- pop the frame when the function returns

"frame" or "stack frame": a data structure that keeps track of variables in the function body, and their values, between the call to the function and its return

* "runtime system" = the code that Python executes to make everything work at runtime

How recursion works

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


>>> fact(4)
24
```

Python's runtime system*
maintains a stack:

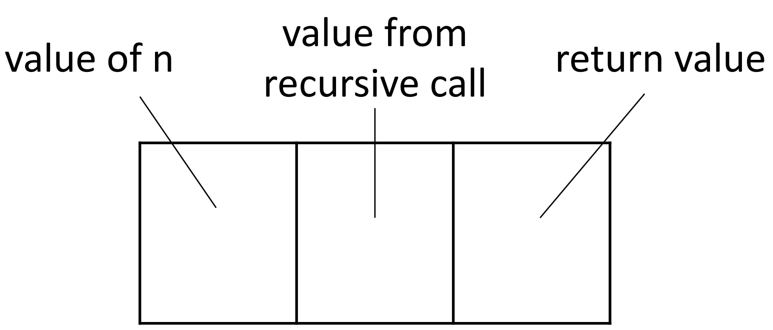
- push a "frame" when a function is called
- pop the frame when the function returns

sometimes called the "runtime stack"

* "runtime system" = the code that Python executes to make everything work at runtime

How recursion works

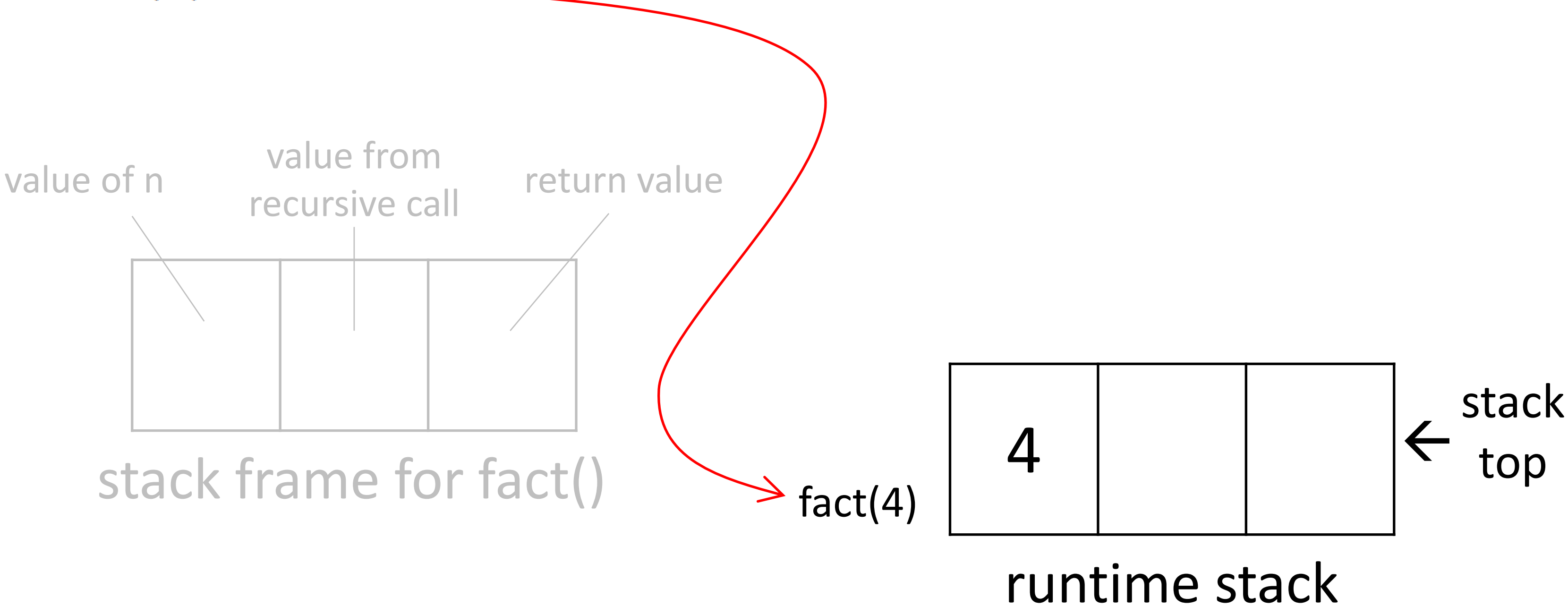
```
>>> def fact(n):
        if n == 0:
            return 1
        else:
            return n * fact(n-1)


>>> fact(4)
24
```



stack frame for fact()

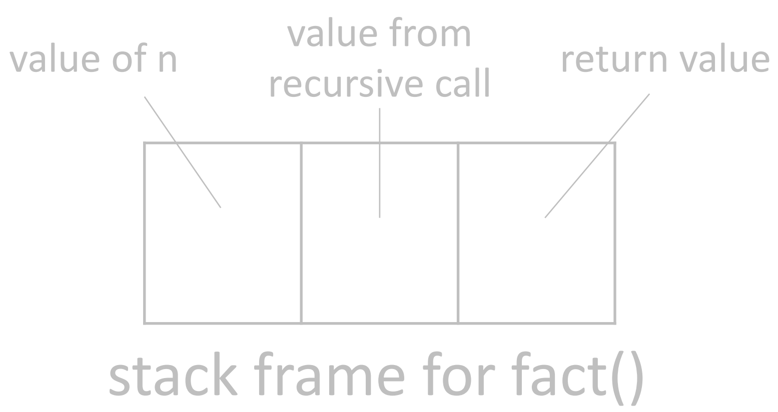
How recursion works

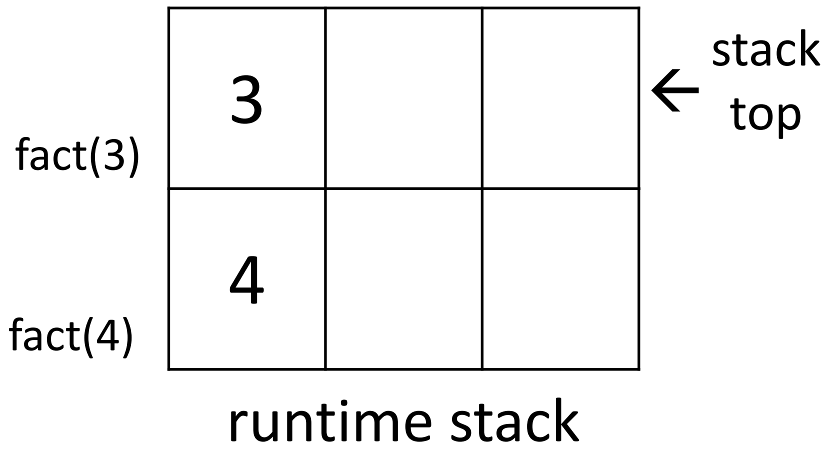
```
>>> def fact(n):
        if n == 0:
            return 1
        else:
            return n * fact(n-1)


>>> fact(4)
24
```



How recursion works

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


>>> fact(4)
24
```

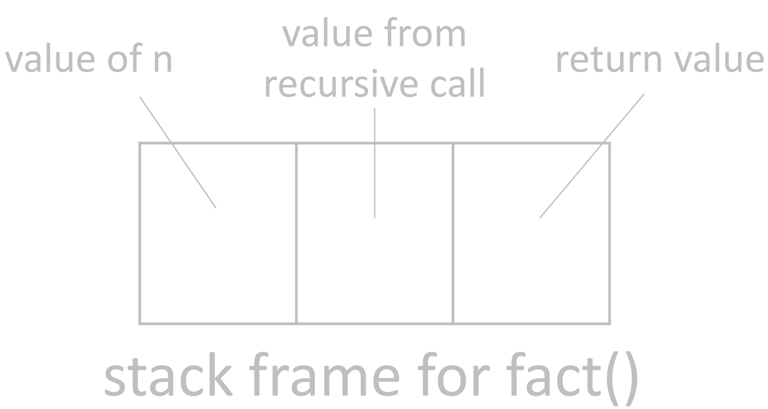


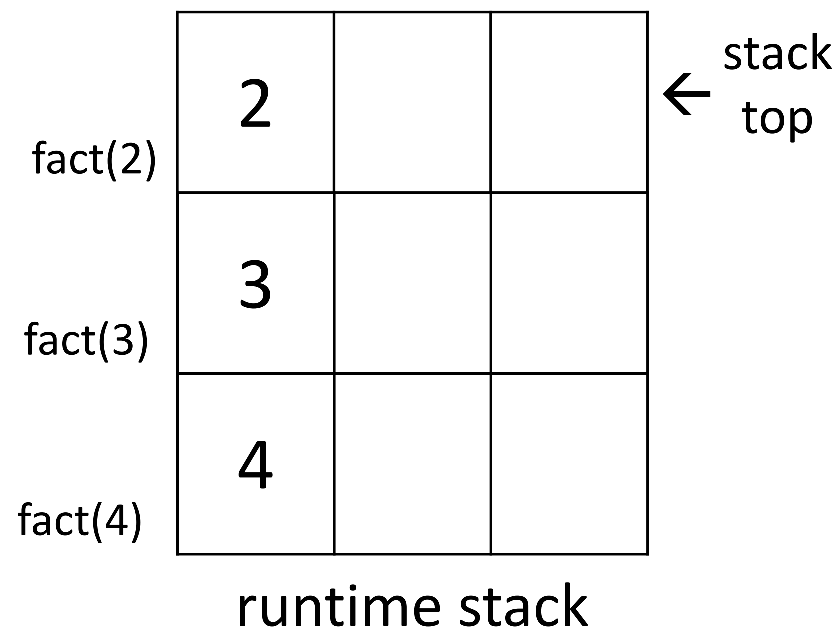


How recursion works

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

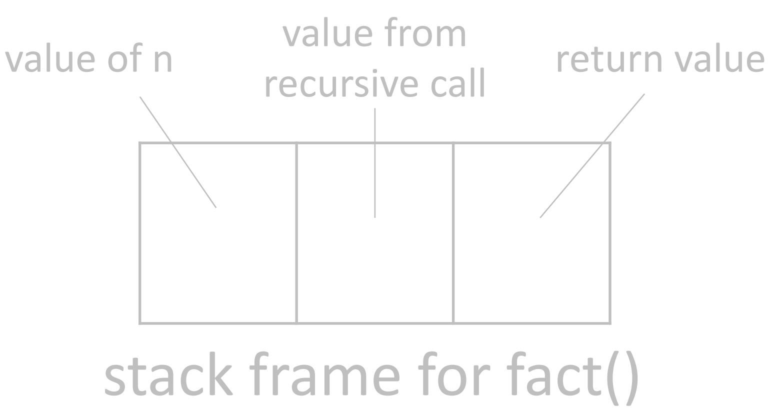
```
>>> fact(4)
24
```

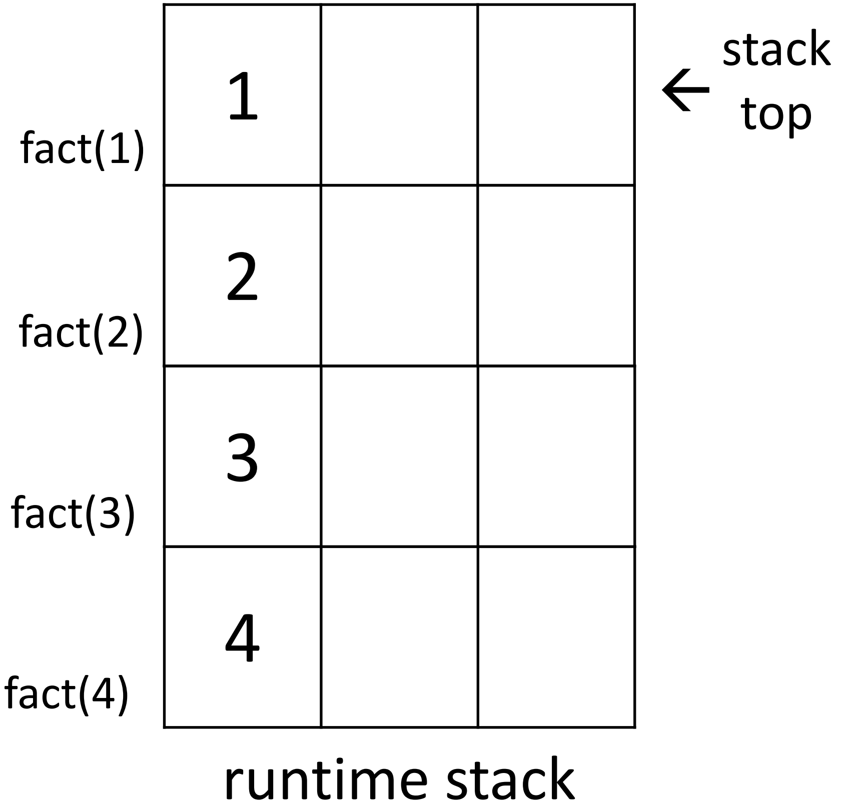




How recursion works

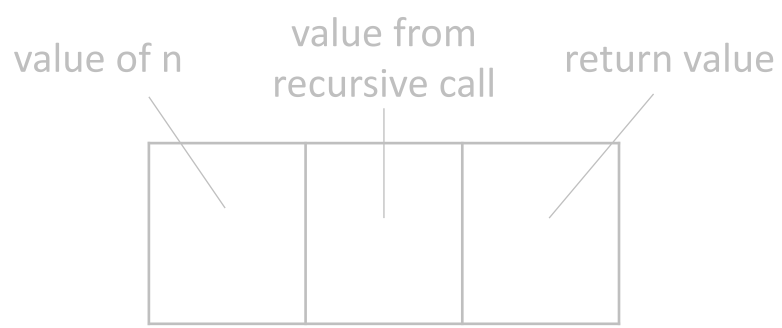
```
>>> def fact(n):
        if n == 0:
            return 1
        else:
            return n * fact(n-1)

>>> fact(4)
24
```



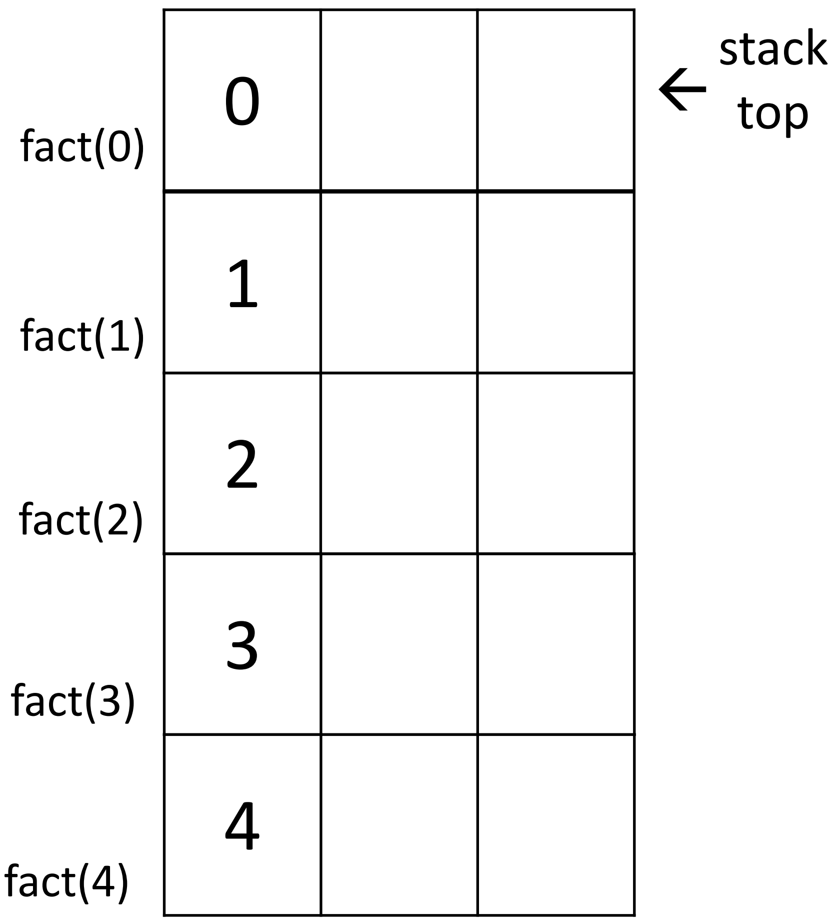


How recursion works

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


>>> fact(4)
24
```



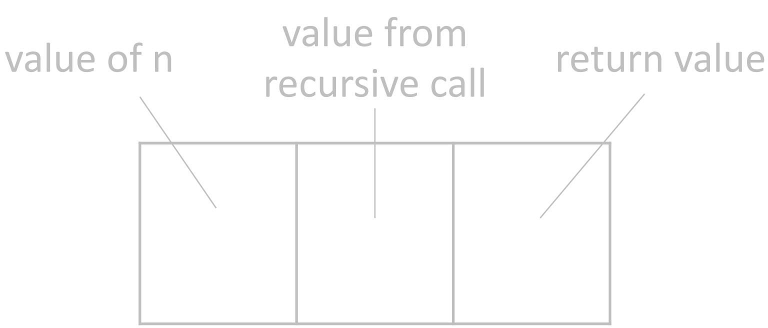
stack frame for fact()



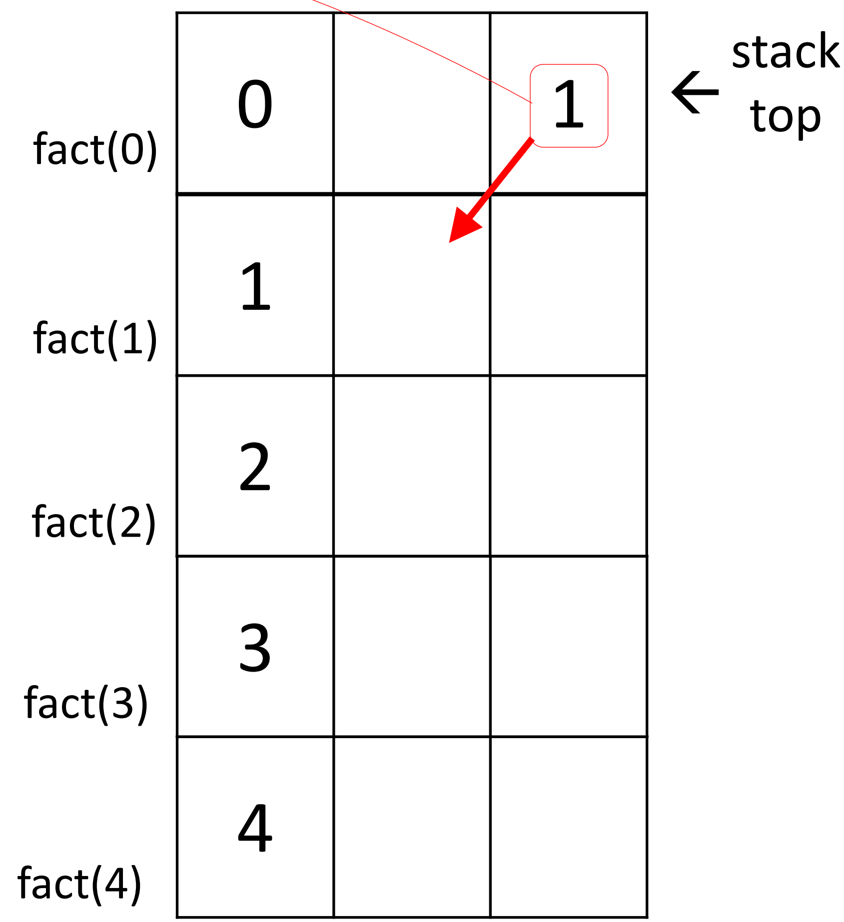
runtime stack

How recursion works

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


>>> fact(4)
24
```



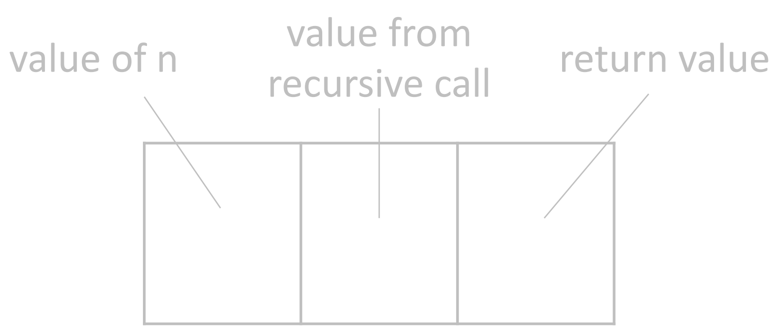
stack frame for fact()



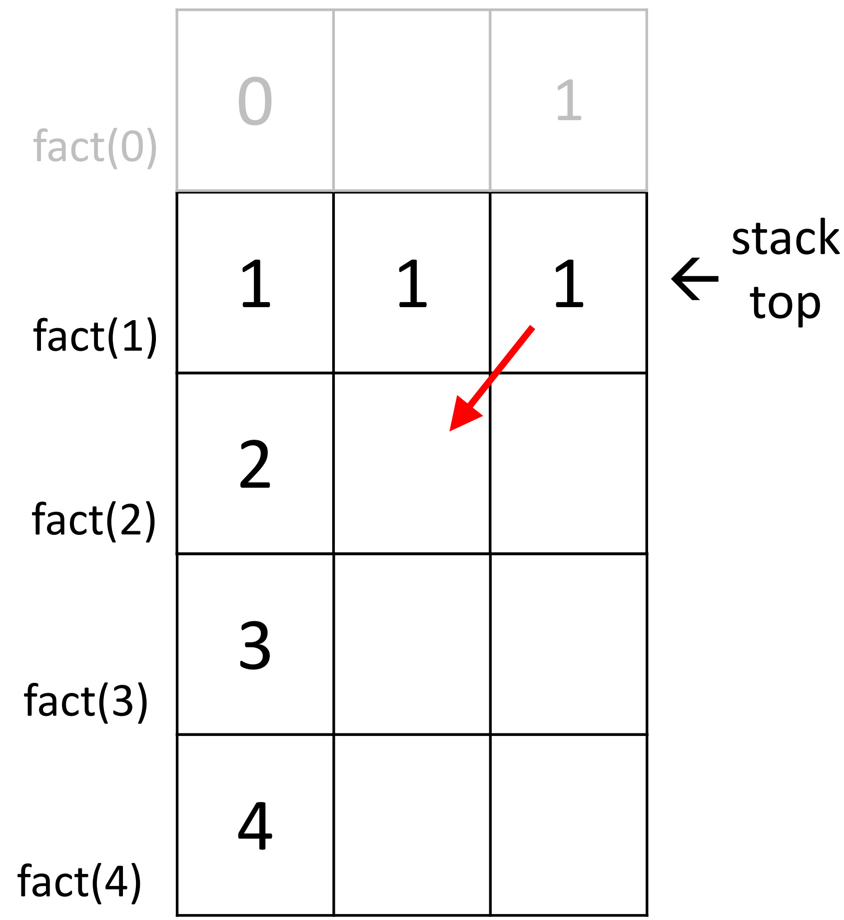
runtime stack

How recursion works

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


>>> fact(4)
24
```



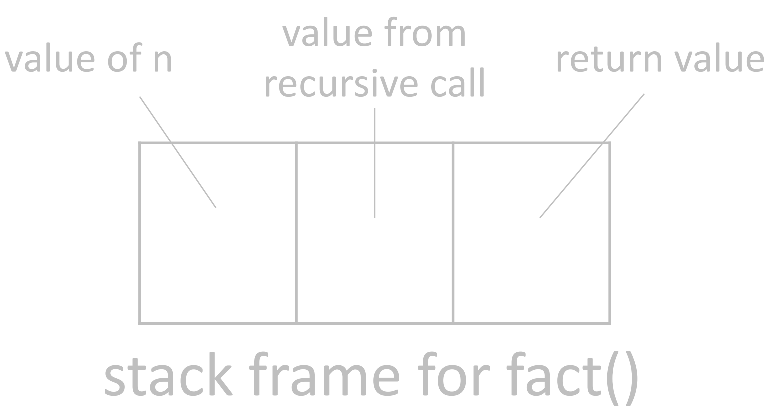
stack frame for fact()

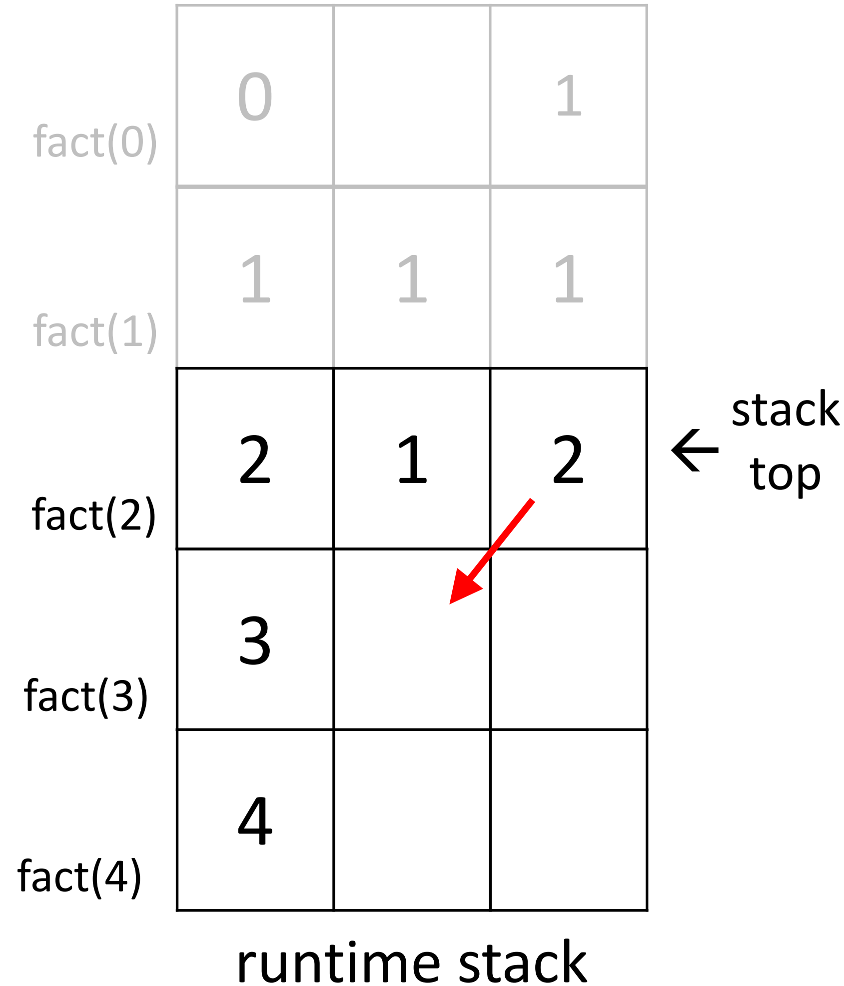


runtime stack

How recursion works

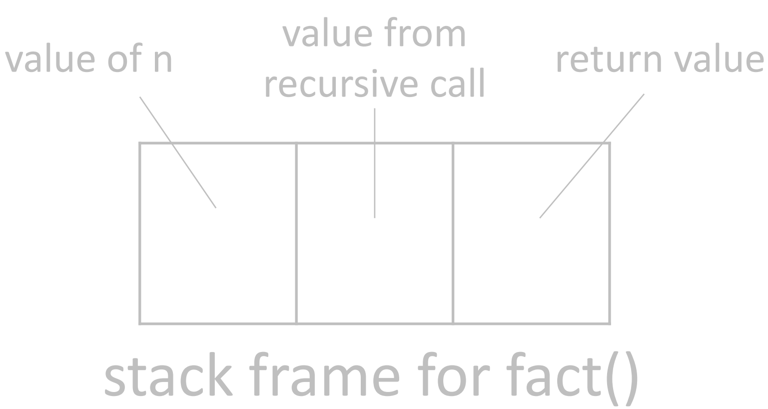
```
>>> def fact(n):
        if n == 0:
            return 1
        else:
            return n * fact(n-1)


>>> fact(4)
24
```

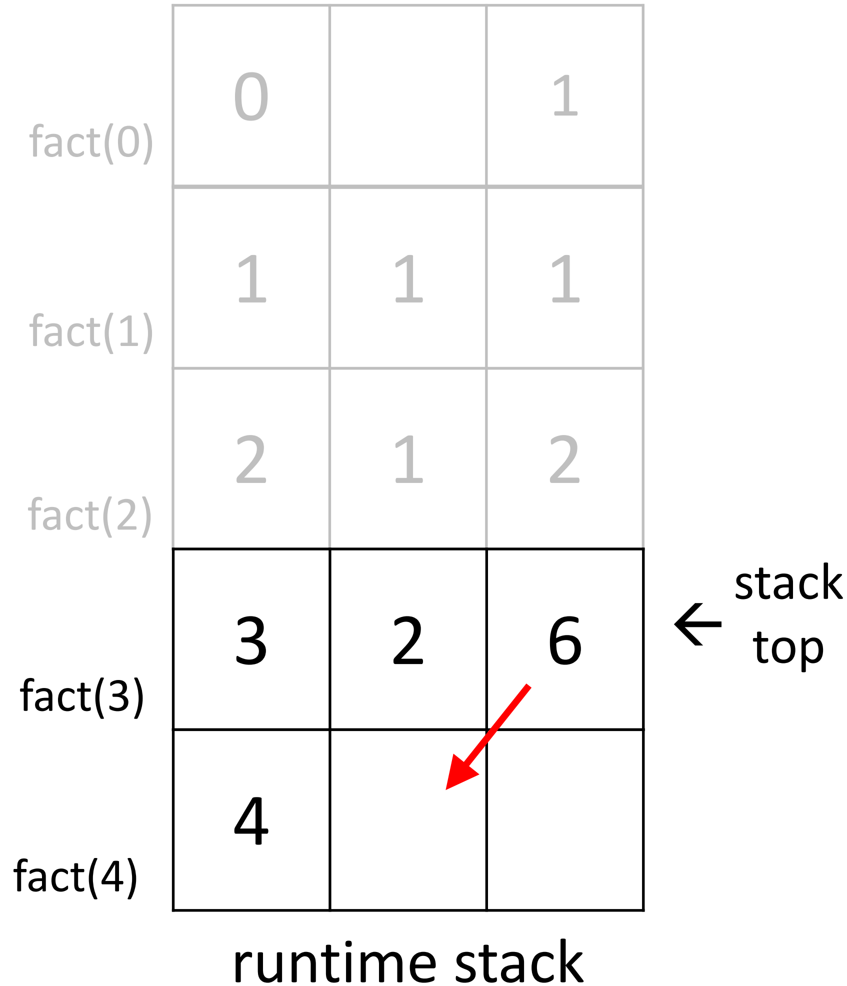




How recursion works

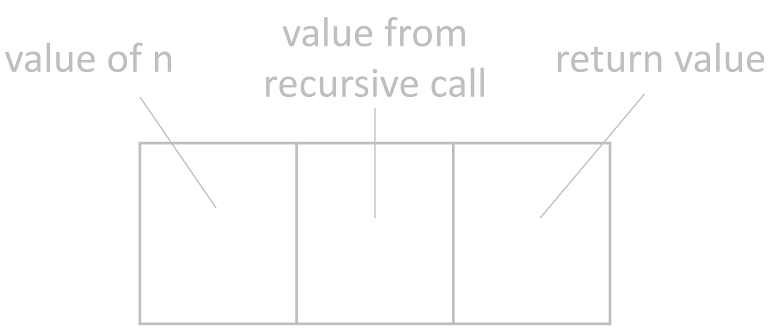
```
>>> def fact(n):
        if n == 0:
            return 1
        else:
            return n * fact(n-1)


>>> fact(4)
24
```



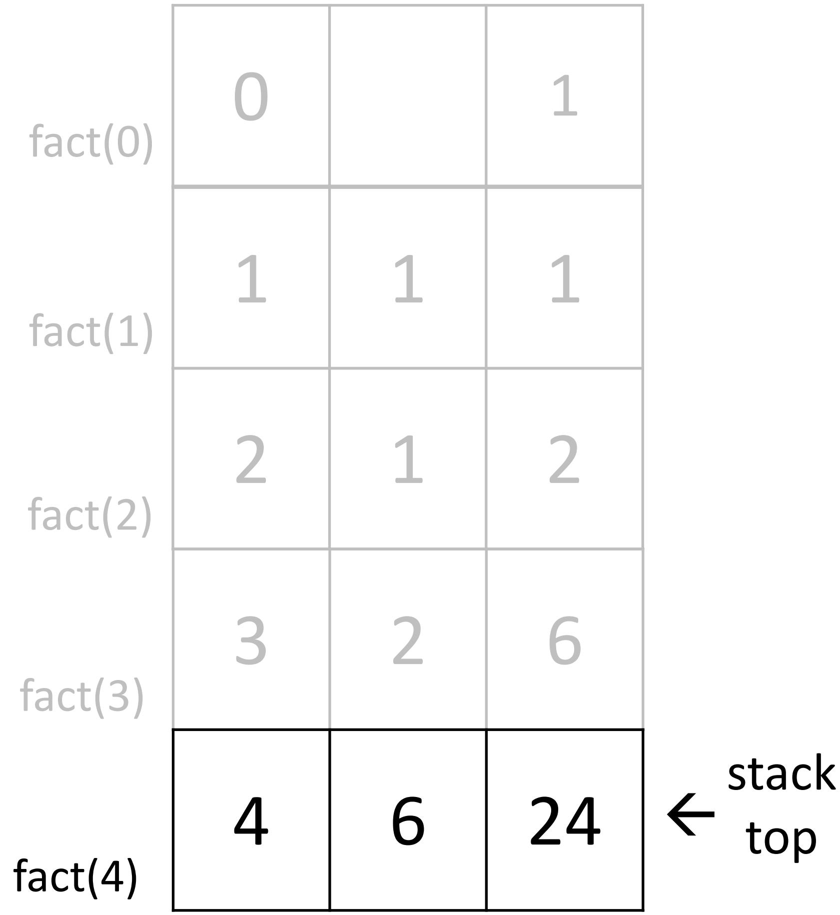


How recursion works

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


>>> fact(4)
24
```



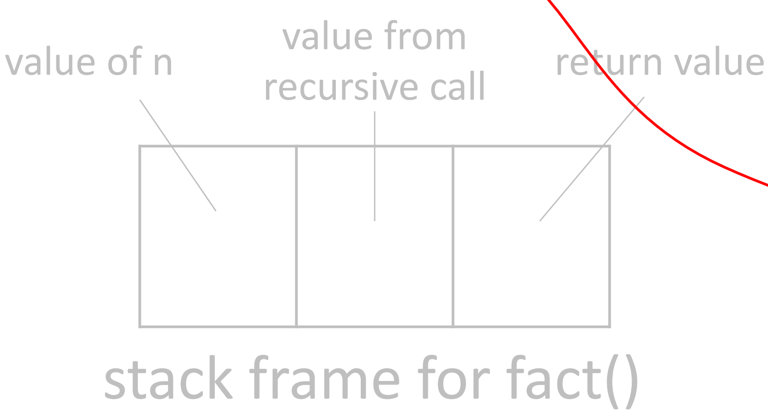
stack frame for fact()

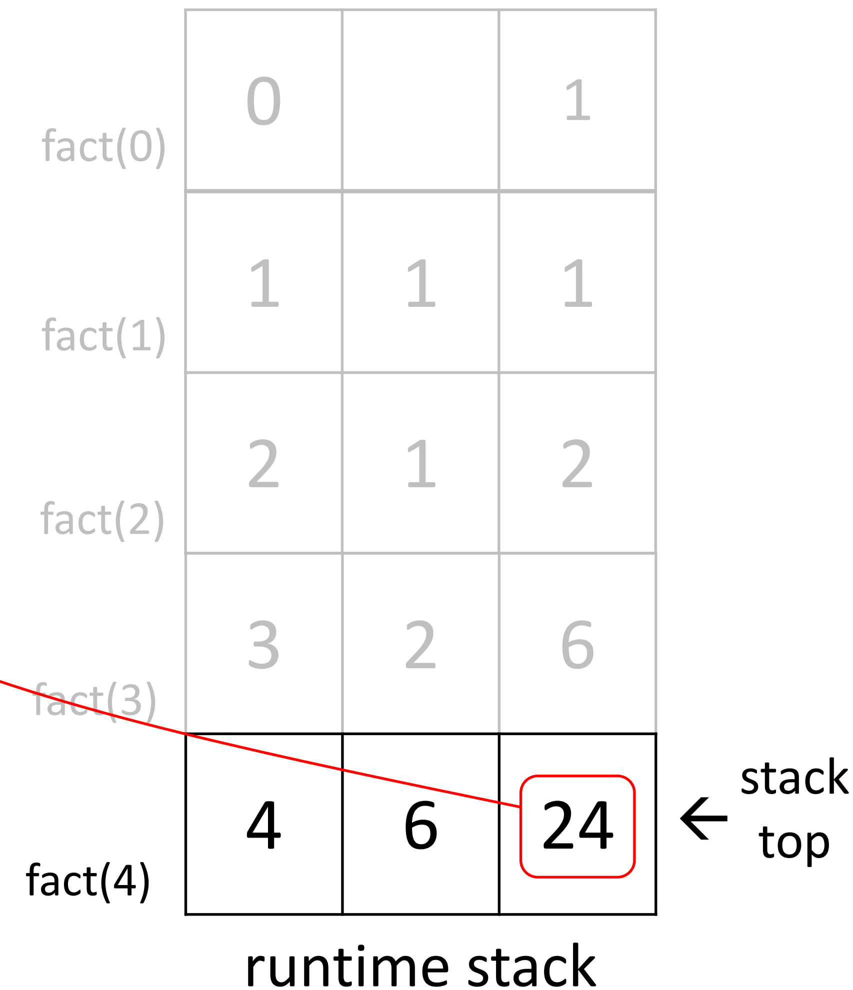


runtime stack

How recursion works

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

>>> fact(4)
24
```





The runtime stack

- The use of a *runtime stack* containing *stack frames* is not specific to recursion
  - all function and method invocations use this mechanism
  - not just in Python, but other languages as well (Java, C, C++, …)

Problem 5

Write a recursive function to print the numbers from 1 through n, one per line.

Usage:

```
>>> print_n(6)
1
2
3
4
5
6
```

Solution

```
def print_n(n):
    if n == 0:
        return
    else:
        print_n(n-1)
        print(n)
```

base case: n = 0

recursive case: n != 0

recursive call is on a smaller value

Recursion How to

To write a recursive function, figure out:

What values are involved in the computation?

- these will be the arguments to the recursive function

- *Base case(s)*
  - when does the recursion stop?
  - what is the simple value or data that can be computed and returned?
- *Recursive case(s)*
  - what is the "smaller problem" to pass to the recursive call?
  - what does a single round of computation involve?

EXERCISE-ICA-19

Do all problems 3-5.

Recursion

- Except for print_n(n), the recursive solutions have all followed a similar pattern to sumlist() below:

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

- Let’s look at the solution for replace() (similar pattern)

Recursion

- Replace

```
def replace(s, a, b):
    if s == "":
        return ""
    elif s[0] == a:
        return b + replace(s[1:], a, b)
    else:
        return s[0] + replace(s[1:], a, b)
```

- Small modification: What if **a** is more than one character long?

EXERCISE ICA-20 p. 1-2

Problem 1: replace(s, a, b) where a is a string of arbitrary length

Problem 2: get_even_positions(alist)

(The recursive solutions follow the same pattern that we have seen so far.)

Recursion

- For some problems, we need to follow a different pattern:
  - recurse and get the result of the recursive case
  - return a value based on that answer

Recursion

- For some problems, we need to follow a different pattern:
  - recurse and get the result of the recursive case
  - return a value based on that answer

```
def my_function(arg):
    if <expr1>:
        return <some value>
    result = my_function(arg[1:])
    if <condition based on result> :
        return <expr2>
    else:
        return <expr3>
```

EXERCISE ICA-20 p. 3

Write a function `max_l(alist)` that returns the largest value in `alist`. Assume `alist` has at least one element.

Usage:

```
>>> max_l([8, 3, 24, 7, 9])
24
```

Use the "new" pattern to help write the solution

max_l() solution

Let’s do this on the ELMO.

EXERCISE ICA-20 p. 4

Write the function `maxmin(alist)` described in the ICA.

Note: this returns a tuple!

Think carefully about the base case.

Use the "new" pattern to help write the solution

(Do problem 5 if you finish.)

Versions of sumlist

Version 1	

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

base case

recursive case

One round of computation adds L[0] and the result of the recursive call

argument to recursive call is "rest of the list" after L[0] (recurses on a smaller problem)

Versions of sumlist

Version 2
(variation on version 1)

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

argument to recursive call is "rest of the list" up to the last element (recurses on a smaller problem)

add the last element of L

Versions of sumlist

Version 2 (variation on version 1)	Version 3 ("smaller" need not be by just 1)

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

```
def sumlist(L):
    if len(L)  = 0:
        return 0
    elif len(L) == 1:
        return L[0]
    else:
        return sumlist(L[ :len(L)//2]) + \
               sumlist(L[len(L)//2: ])
```

better for parallel execution

argument to each recursive call is half of the current list (recurses on a smaller problem)

sumlist

```
def sumlist(L):
    if len(L)  = 0:
        return 0
    elif len(L) == 1:
        return L[0]
    else:
        return sumlist(L[ :len(L)//2]) + sumlist(L[len(L)//2: ])
```

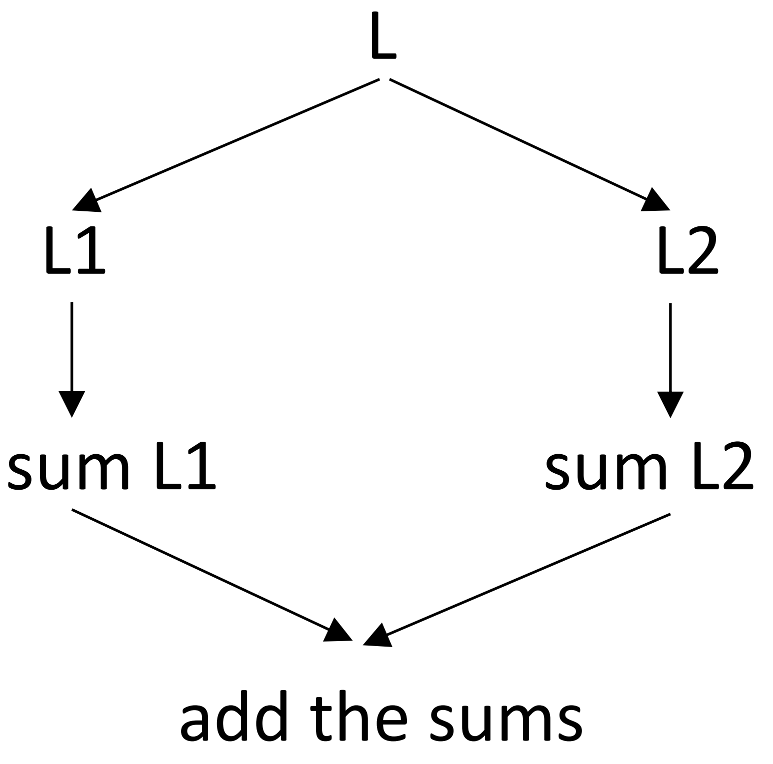
recursive sumlist

input list

split into two halves

add the halves (recursively)

return the sum of the sums

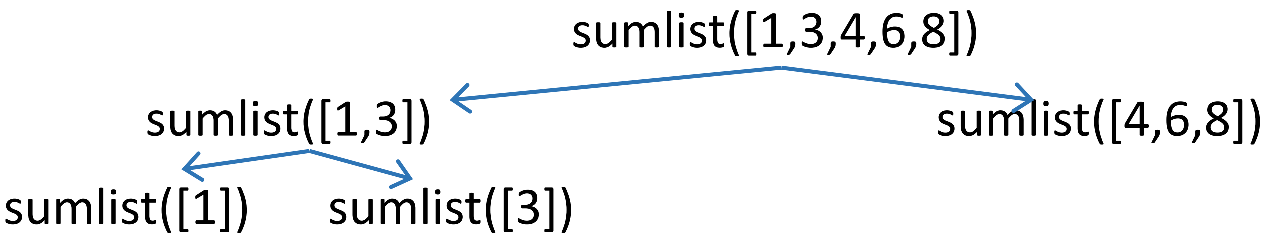


sumlist: example

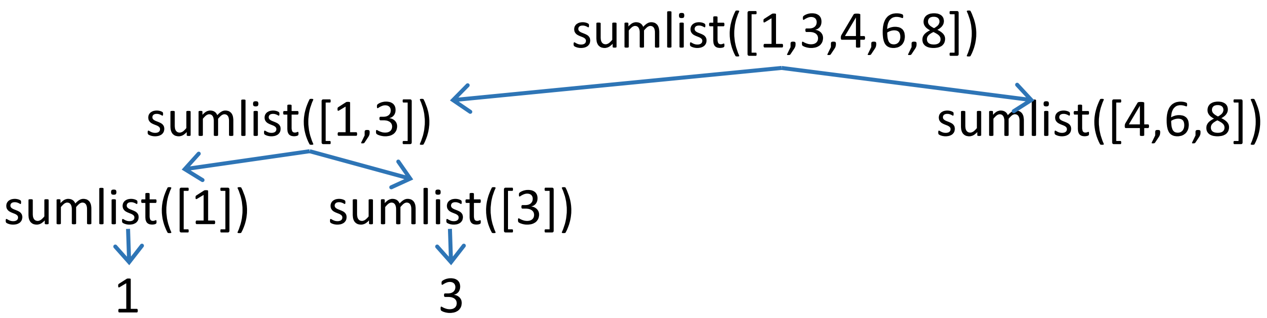
```
sumlist([1,3,4,6,8])
```

sumlist: example

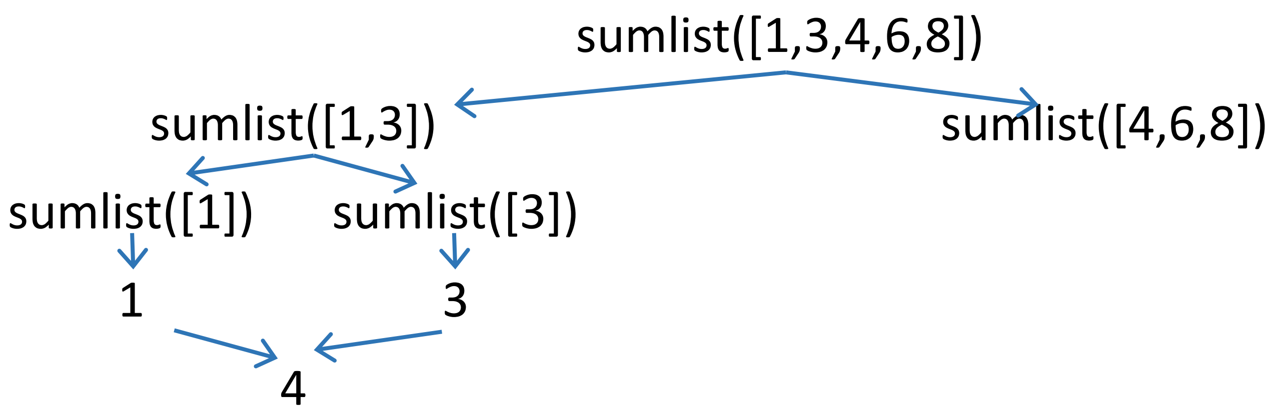
sumlist: example



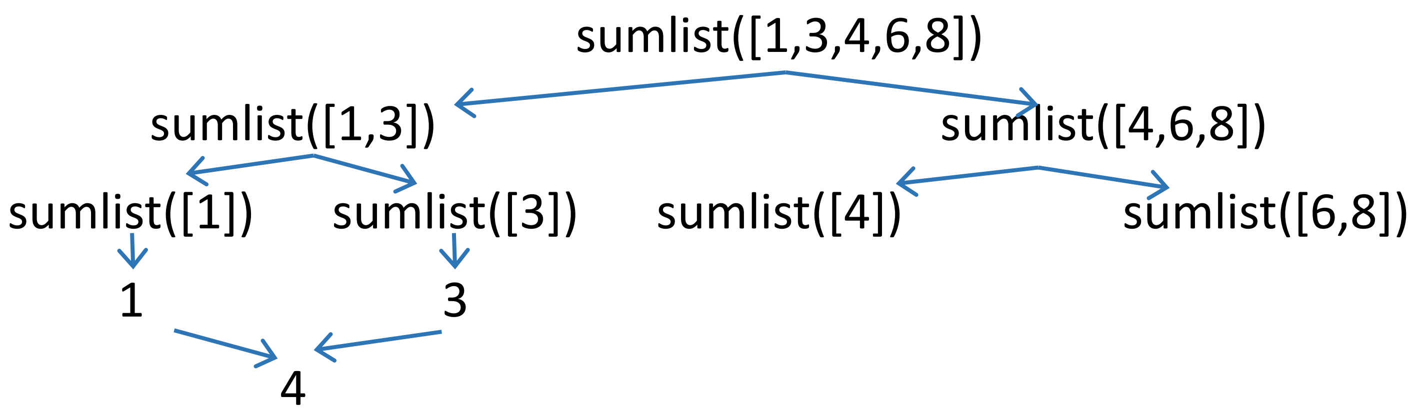
sumlist: example



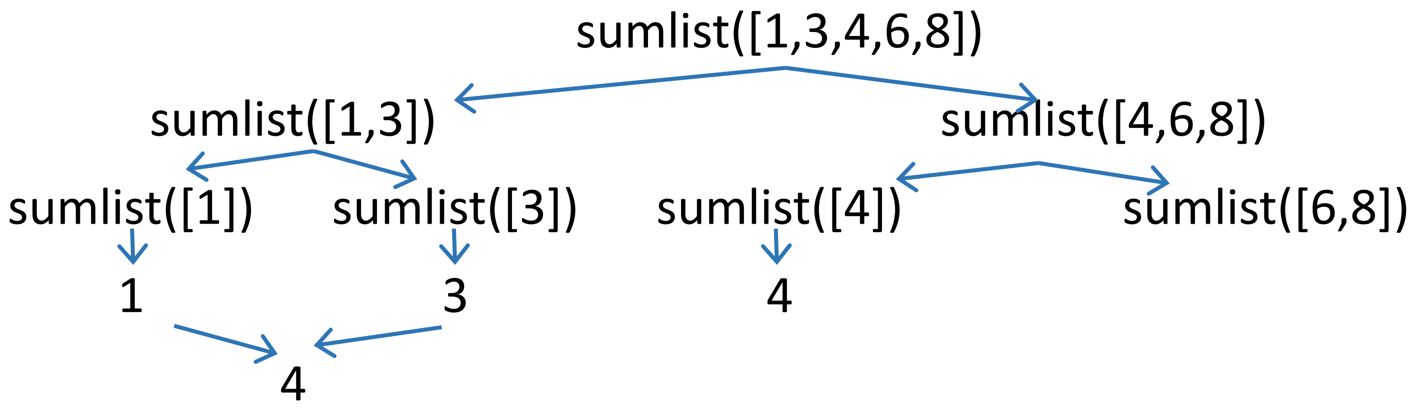
sumlist: example



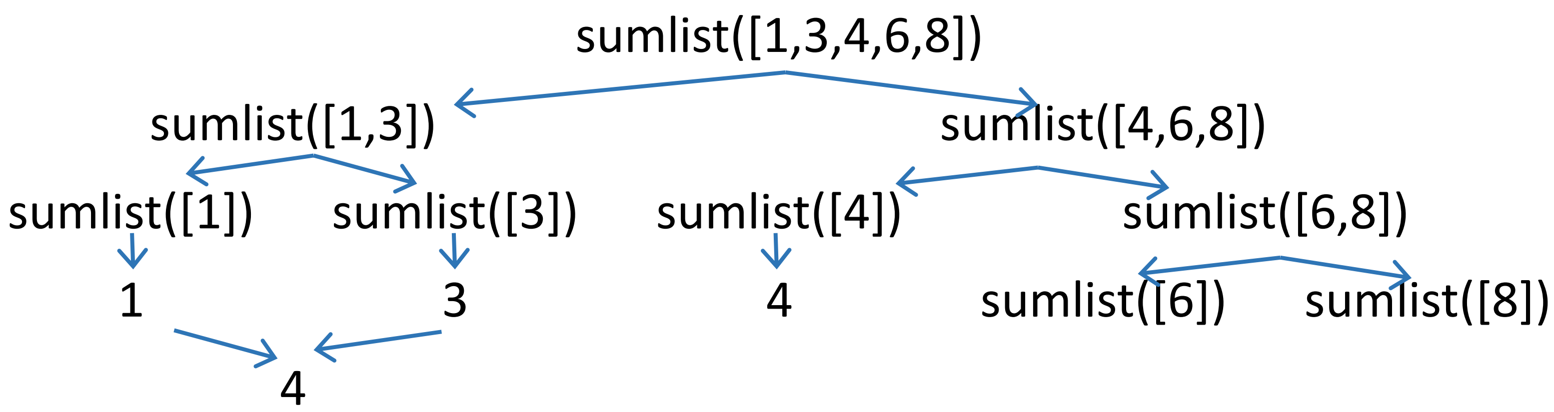
sumlist: example



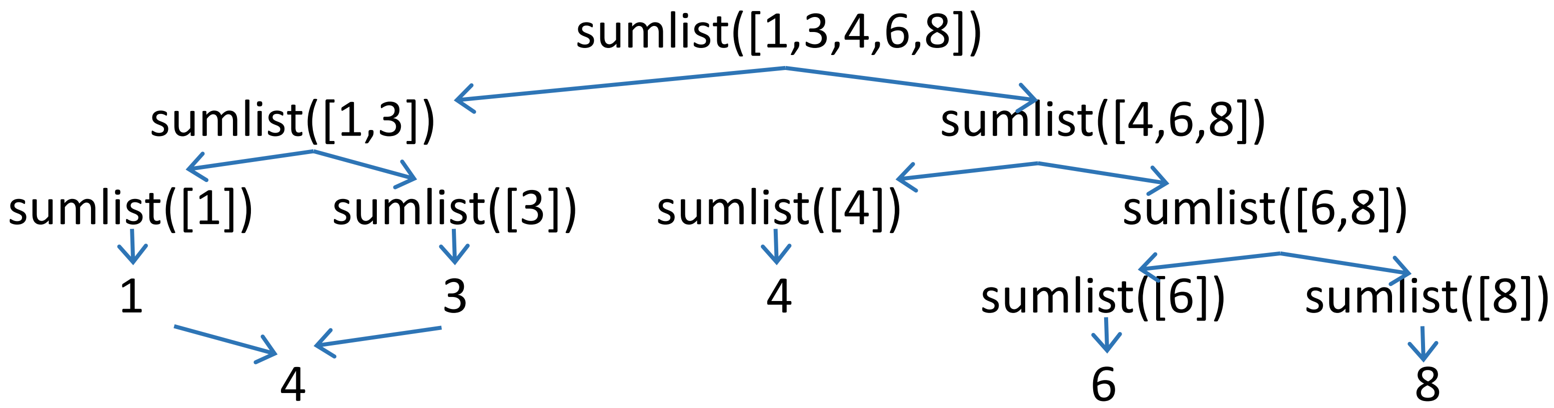
sumlist: example



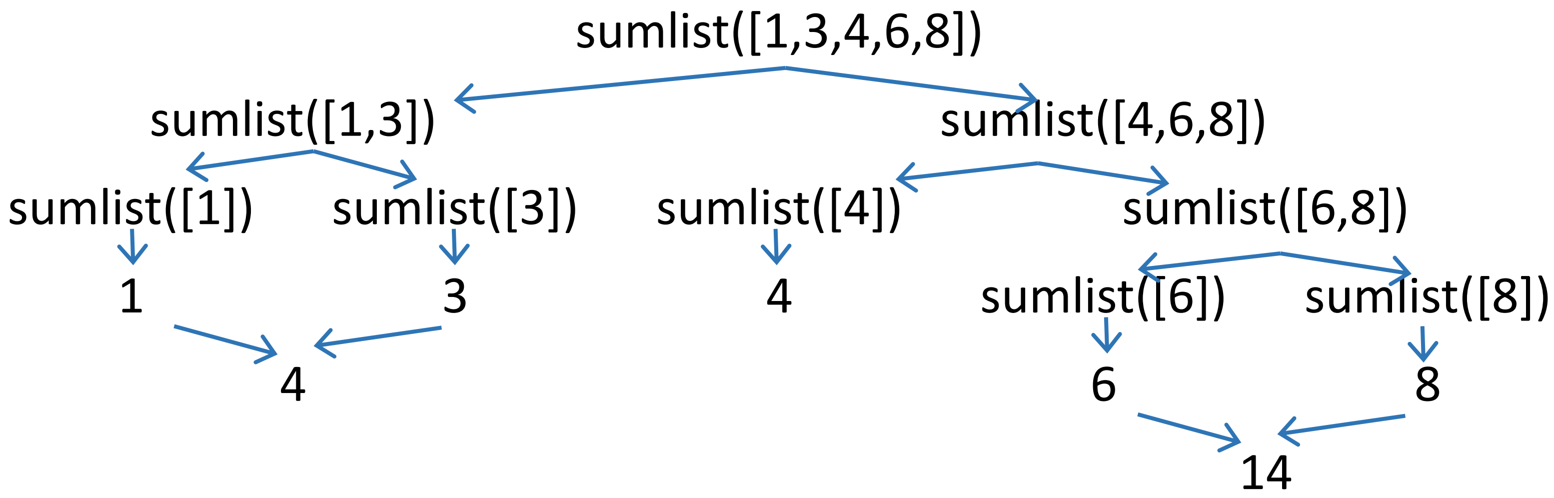
sumlist: example



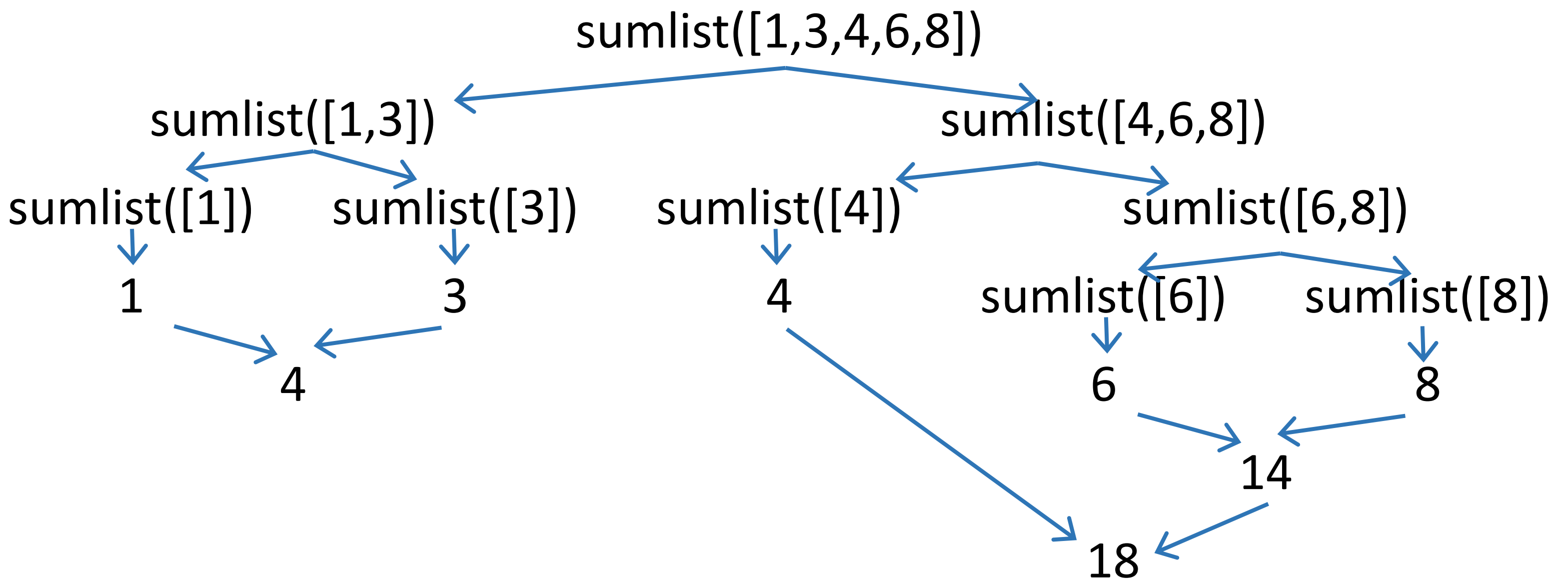
sumlist: example



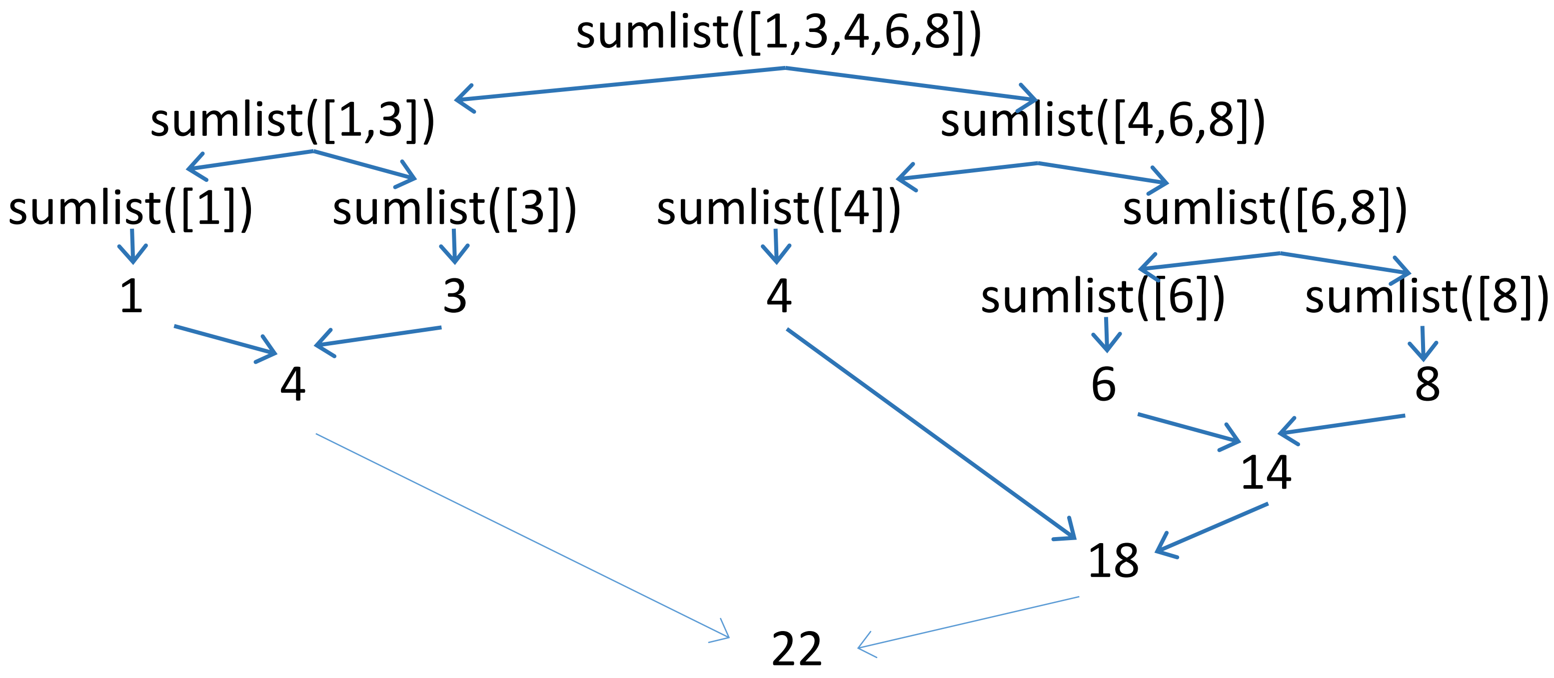
sumlist: example



sumlist: example



sumlist: example



EXERCISE ICA-21 p. 1

Write a sumlist() two different ways.

recursion: example search

Searching an unsorted list

- Problem: Given an unsorted list *L* and a value *a*, determine whether or not *a* is in *L*.

a 29

	0	1	2	3	4	5	6	7	8	9
L	3	81	2	14	64	12	8	29	14	5

Searching an unsorted list

- Problem: Given an unsorted list *L* and a value *a*, determine whether or not *a* is in *L*.

a 29

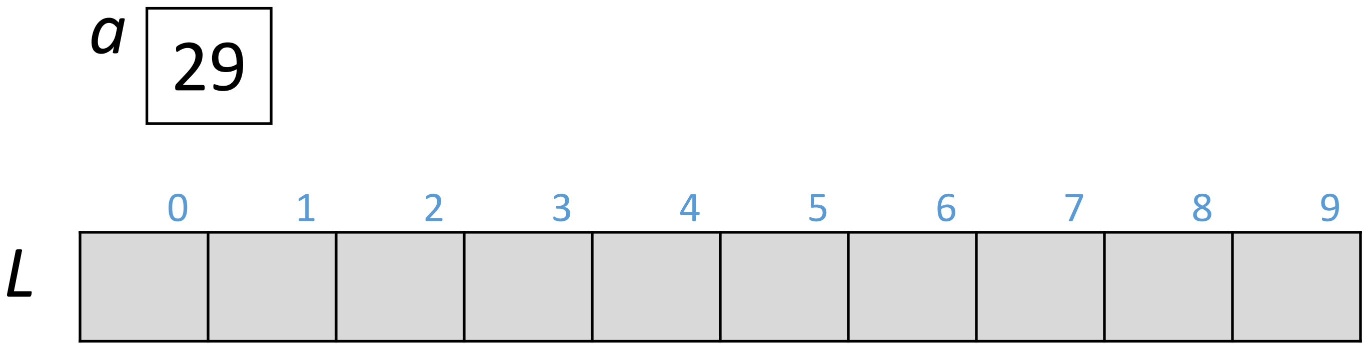
L	0	1	2	3	4	5	6	7	8	9
	3	81	2	14	64	12	8	29	14	5

↑

- Linear search: sequentially look at (possibly) all values in the list.

Searching a sorted list

- Problem: Given a sorted list *L* and a value *a*, determine whether or not *a* is in *L*.



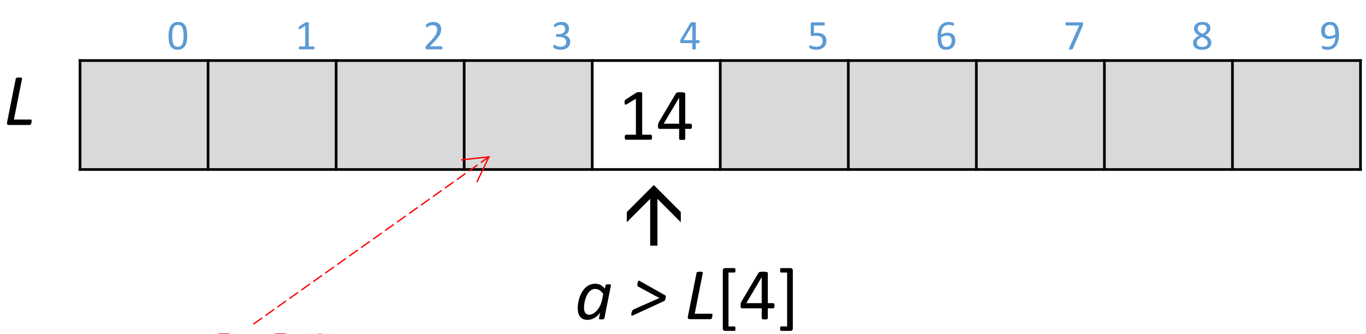
Searching a sorted list

- Problem: Given a sorted list *L* and a value *a*, determine whether or not *a* is in *L*.

a 29

pick a value *i* in range(len(*L*))

say, *i* = 4



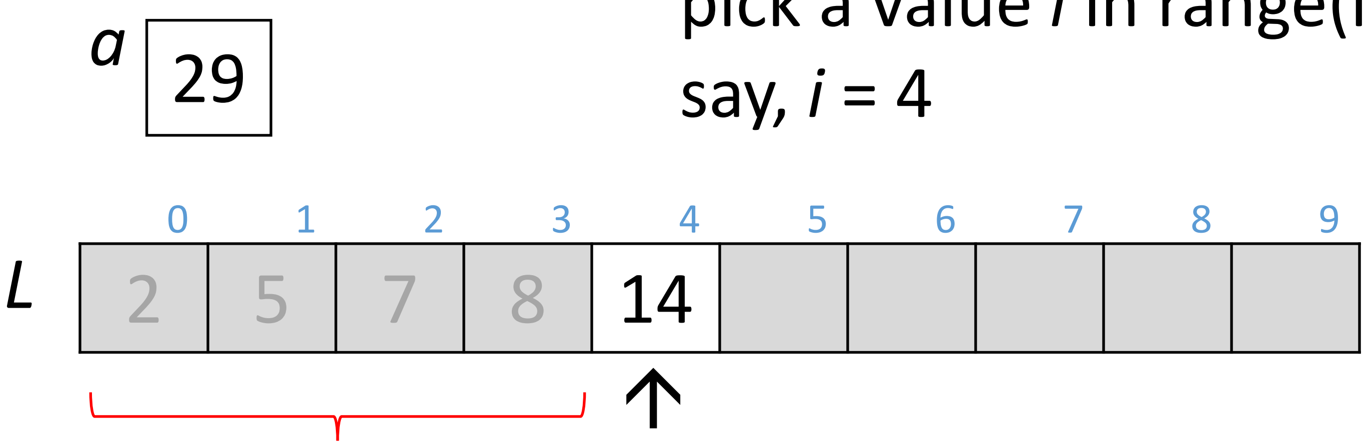
Q: Can *L*[3] be *a*?

Searching a sorted list

- Problem: Given a sorted list *L* and a value *a*, determine whether or not *a* is in *L*.

pick a value *i* in range(len(*L*))

say, *i* = 4



a > *L*[4]

L sorted and *a* > L[4]
means *a* cannot be any
of these elements

Binary search: recursive solution

binary search - find an item in a sorted list

```
if the list is empty
    the item is not found (return False)
look at the middle of the list
if we found the item
    then done (return True)
else
    if the item is less than the middle
        search in the lower half of the list
    else
        search in the upper half of the list
```

EXERCISE-ICA21 p. 2

Write a recursive function bin_search(alist, item) that that searches for item in alist and returns True if found and False otherwise.

Usage:

```
>>>bin_search([4, 25, 28, 33, 47, 54, 65, 83], 65)
True
>>>
```

Binary search

```
def bin_search(L, item):
    if L == []:
        return False
    mid = len(L)//2
    if L[mid] == item :
        return True
    if item < L[mid]:
        return bin_search(L[0:mid], item)
    else:
        return bin_search(L[mid+1:], item)
```

recursion: example

Example: merging two sorted lists

Problem: Given two sorted lists L1 and L2, merge them into a single sorted list (recursively)

Example: L1 = [11, 22, 33], L2 = [5, 10, 15]

- Output: [5, 10, 11, 15, 22, 33]
  - can't just concatenate the lists
  - can't alternate between the lists

Merging: values involved

Problem: Given two sorted lists L1 and L2, merge them into a single sorted list

1. Values involved in the computation in each (recursive) call ?

   L1 and L2

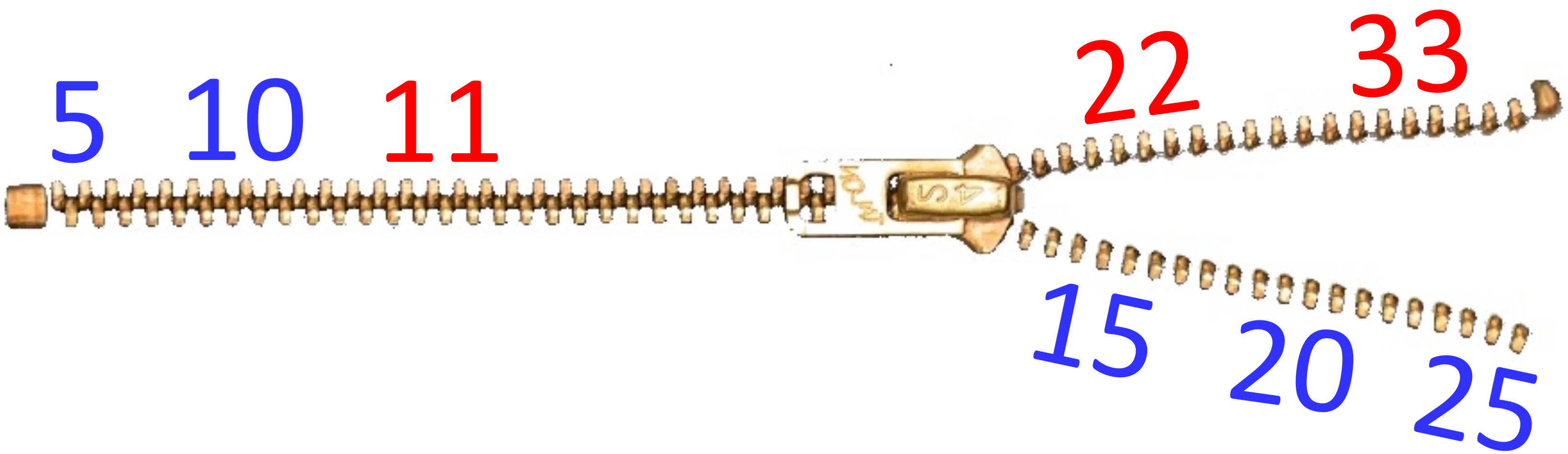
So the recursive function will look something like

```
def merge(L1, L2):   # may need another parameter
    …
```

Merging: repetition

Problem: Given two sorted lists L1 and L2, merge them into a single sorted list

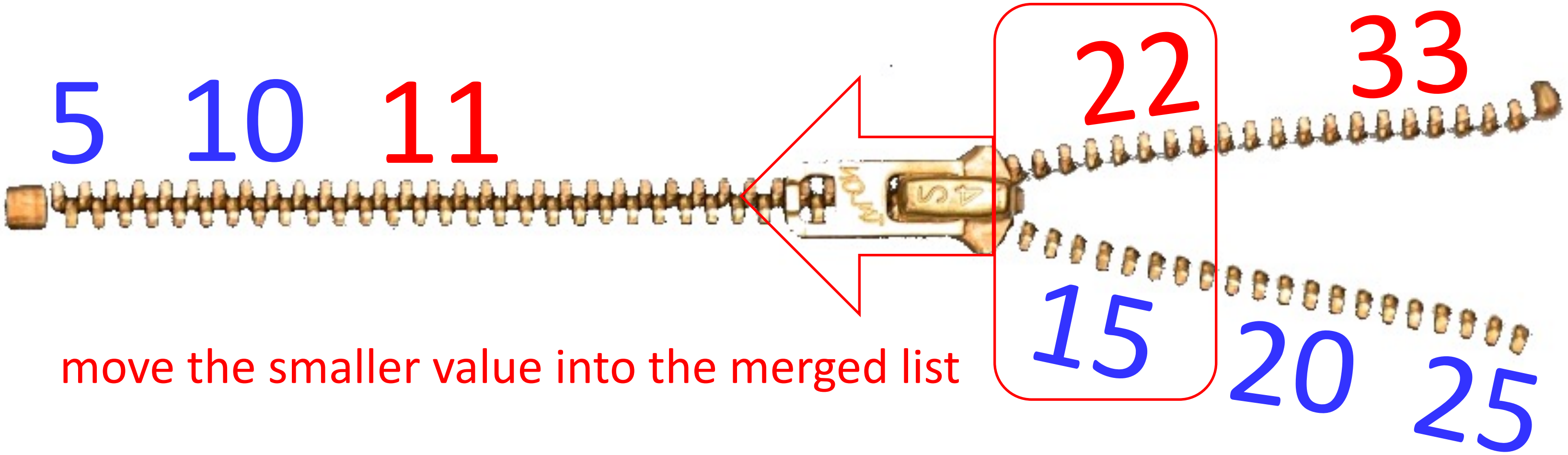
2. What does the computation involve in each call?



Merging: repetition

Problem: Given two sorted lists L1 and L2, merge them into a single sorted list

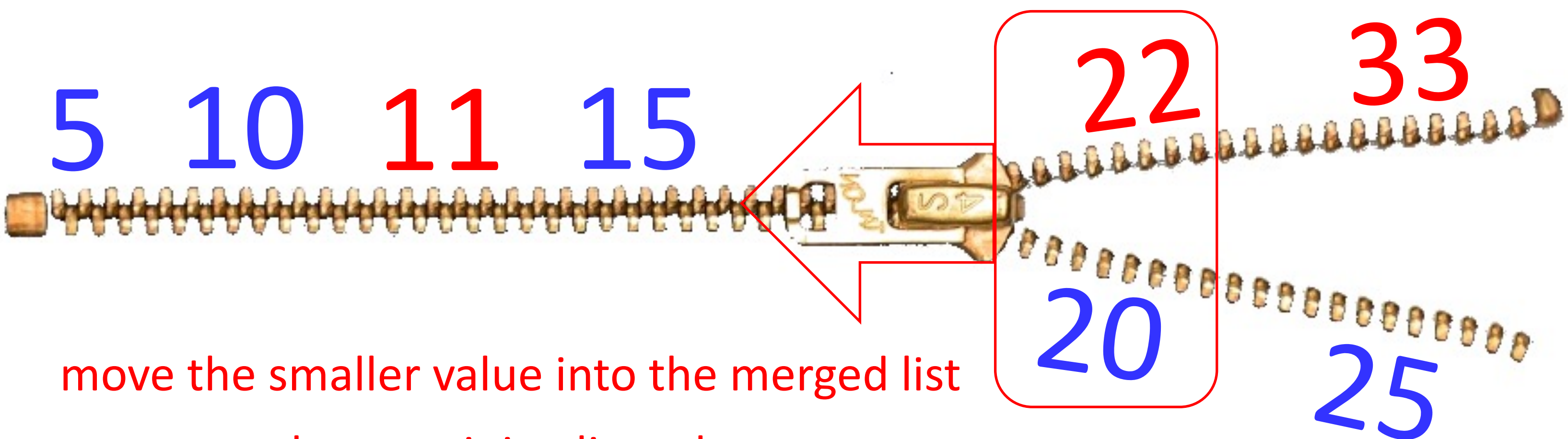
2. What does the computation involve in each call?



Merging: repetition

Problem: Given two sorted lists L1 and L2, merge them into a single sorted list

2. How does the problem (or data) get smaller?



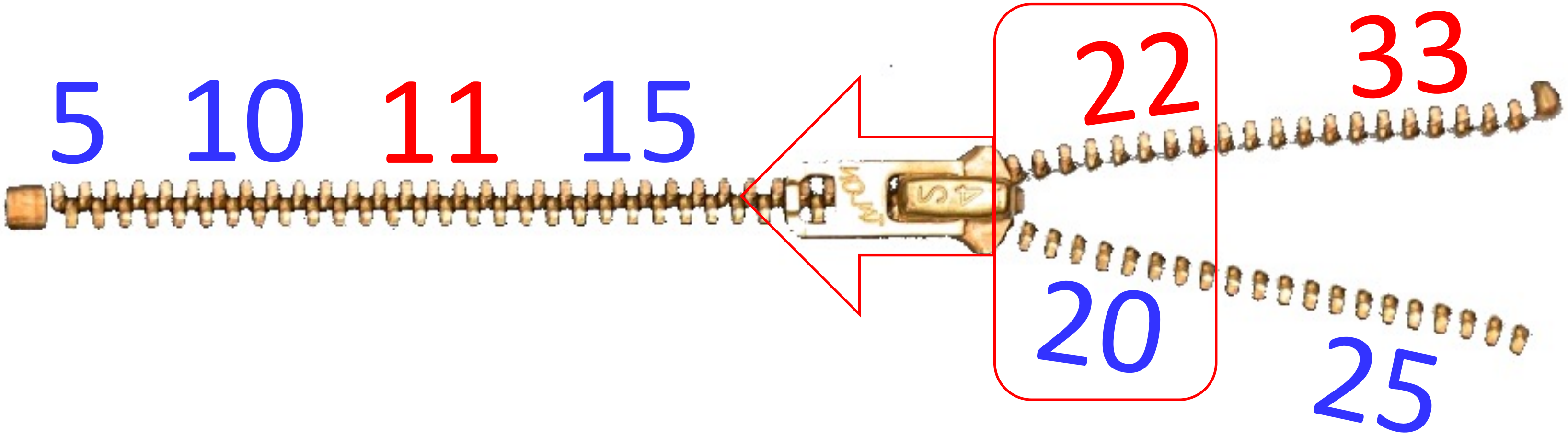
move the smaller value into the merged list

repeat on the remaining list values

Merging: base case

Problem: Given two sorted lists L1 and L2, merge them into a single sorted list

3. When can't we make the data smaller?

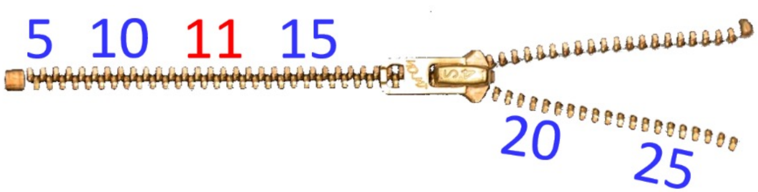


Merging: base case

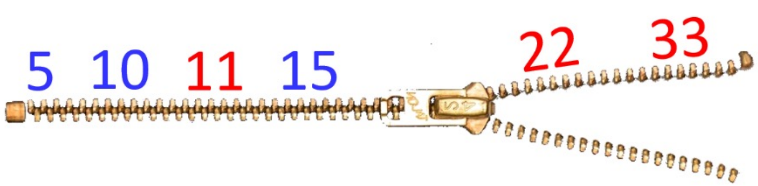
Problem: Given two sorted lists L1 and L2, merge them into a single sorted list

3. When can't we make the data smaller?

– when either L1 or L2 is empty



or



in this case, concatenate the other list into the merged list

Merging: base case

The code looks something like:

```
def  merge(L1, L2, merged):  # note the new parameter
        if L1 == []:
              return merged + L2
        elif L2 == []:
              return merged + L1
        else:
              ….
```

Call it like this:

merge([11,22,33], ([5,10,15],[])

Merging: base case

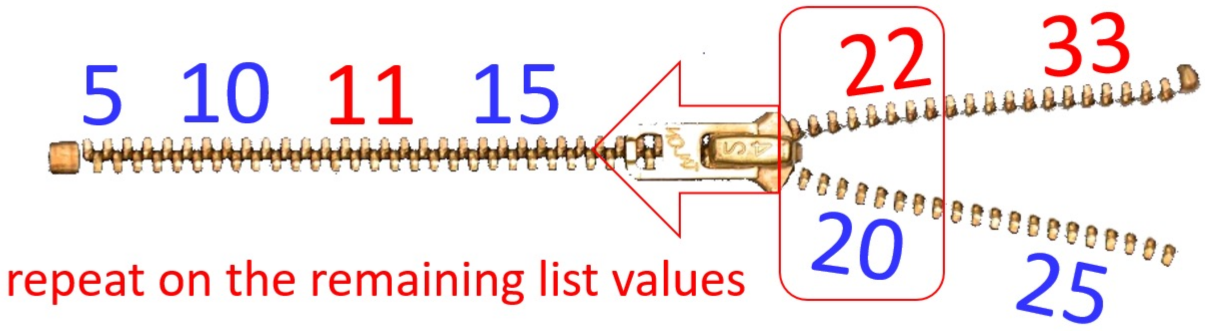
We can combine the two base cases:

```
def  merge(L1, L2, merged): # note the new parameter
    if L1 == [] or L2 == []:
        return merged + L1 + L2
    else:
        ….
```

Merging: recursive case

Problem: Given two sorted lists L1 and L2, merge them into a single sorted list

4. What is "the rest of the computation"?
   - "repeat on the remaining list values"



EXERCISE

Given the pseudocode below, write the recursive cases for merge.

The arguments to merge are lists L1, L2, and merged

```
if L1[0] <= L2[0]
    put L1[0] into the merged list
    recursively merge using the rest of L1, L2, and merged
else
    put L2[0] into the merged list
    recursively merge using L1, the rest of L2, and merged
```

Merging: recursive case –V1

```
if L1[0] < L2[0]:
    merged.append(L1[0])
    return merge(L1[1:], L2, merged)
else:
    merged.append( L2[0] )
    return merge(L1, L2[1:], merged)
```

Merging: recursive case-V2

```
if L1[0] < L2[0]:
        new_merged = merged + [ L1[0] ]
        new_L1 = L1[1: ]
        new_L2 = L2
else:
        new_merged = merged + [ L2[0] ]
        new_L1 = L1
        new_L2 = L2[1: ]
return merge(new_L1, new_L2, new_merged)
```

Merging: putting it all together

```
def  merge(L1, L2, merged):
    if L1 == [] or L2 == []:
        return merged + L1 + L2
    else:
        if L1[0] < L2[0]:
            new_merged = merged + [ L1[0] ]
            new_L1 = L1[1: ]
            new_L2 = L2
        else:
            new_merged = merged + [ L2[0] ]
            new_L1 = L1
            new_L2 = L2[1: ]
        return merge(new_L1, new_L2, new_merged)
```

base case

recursive case

```
>>> def merge(L1,L2,merged):
        if L1 == [] or L2 == []:
                return merged + L1 + L2
        else:
                if L1[0] < L2[0]:
                        new_merged = merged + [L1[0]]
                        new_L1 = L1[1:]
                        new_L2 = L2
                else:
                        new_merged = merged + [L2[0]]
                        new_L1 = L1
                        new_L2 = L2[1:]
                return merge(new_L1, new_L2, new_merged)


>>> merge([11,22,33],[5,10,15,20,25],[])
[5, 10, 11, 15, 20, 22, 25, 33]
>>>
```

recursion: flow of values

Recursion: flow of values

values are computed and passed down as arguments into the recursive call

```
def merge(L1,L2,merged):
    if L1 == [] or L2 == []:
            return merged + L1 + L2
    else:
            if L1[0] < L2[0]:
                    new_merged = merged + [L1[0]]
                    new_L1 = L1[1:]
                    new_L2 = L2
            else:
                    new_merged = merged + [L2[0]]
                    new_L1 = L1
                    new_L2 = L2[1:]
            return merge(new_L1, new_L2, new_merged)
```

Recursion: flow of values

the computation of each round of repetition takes place as values are passed up as return values

```
def merge(L1,L2,merged):
    if L1 == [] or L2 == []:
        return merged + L1 + L2
    else:
        if L1[0] < L2[0]:
            new_merged = merged + [L1[0]]
            new_L1 = L1[1:]
            new_L2 = L2
        else:
            new_merged = merged + [L2[0]]
            new_L1 = L1
            new_L2 = L2[1:]
        return merge(new_L1, new_L2, new_merged)
```

EXERCISE-ICA21 p. 3

Write a recursive function sum_diag(grid) that returns the sum of the diagonal from upper left to bottom right in a grid, i.e., it sums grid[0][0], grid[1][1], and so on.

Usage:

```
>>> sum_diag([[1,2,3], [10,20,30], [100,200,300]])
321
```

EXERCISE-ICA21 p. 3

sum_diag(grid)

Notice:

For a grid = [[1,2,3], [10,20,30], [100,200,300]]

Each time we slice for the recursion, we have one less row (row 0 is always the current row)

How do we know which column to use?

Idea: introduce a new “helper” function with an additional argument:

sum_diag_helper(grid,col)

EXERCISE-ICA21 p. 4 & 5

Write a recursive function zip(a,b), that combines the elements of lists a and b. You will write this in two ways, per the description in the ICA.

EXERCISE-ICA22 p. 1

A recursive function zip(a,b), that combines the elements of lists a and b is provided.

Modify it to use a helper function. (Read the ICA description.)

recursion: application
merge sort

Sorting

- Problem: Given a list L, sort the elements of L into a list sortedL

- Important problem
  - arises in a wide variety of situations
  - many different algorithms, with different assumptions and characteristics
  - we will consider just one algorithm

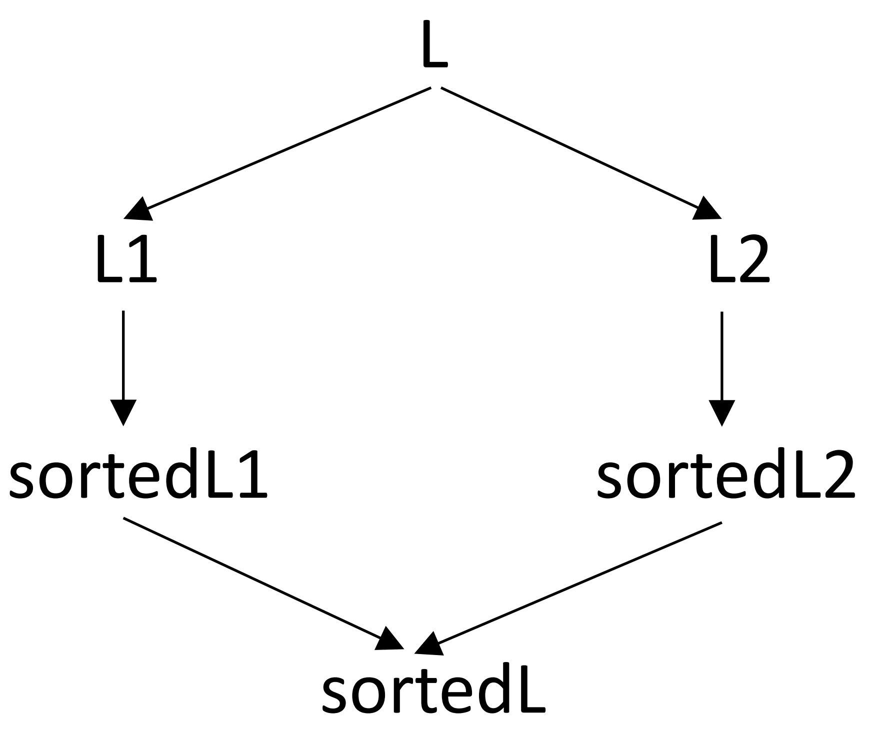
Algorithm: mergesort

input list

split into two halves

sort the halves recursively

merge the sorted lists



Divide and conquer algorithm

Divide and Conquer

- An algorithm paradigm based on multi –branched recursion
  - Recursively break the problem down into two or more sub-problems (until they are trivial to solve)
  - Combine the solutions of the sub-problems to give the solution to the original problem

Mergesort

- Base case: len(L) <= 1
  - no further halving possible
- Recursive case:
  - set up the next round of computation: split the list
  - smaller problem to recurse on: a list of half the size
- Each round of computation: merging the sorted lists
  - has to be done *after* the recursive call

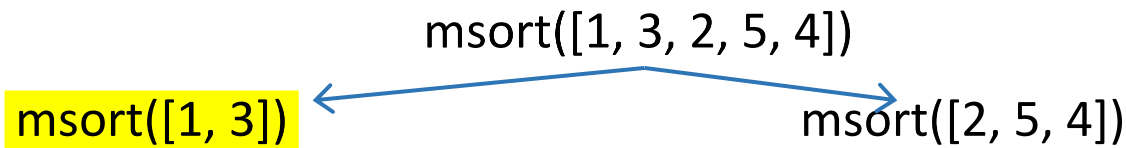
Mergesort

```
def msort(L):
    if len(L) <= 1:
        return L
    else:
        split_pt = len(L)//2
        L1 = L[ :split_pt]
        L2 = L[split_pt: ]
        sortedL1 = msort(L1)
        sortedL2 = msort(L2)
        return  merge(sortedL1, sortedL2,[])
```

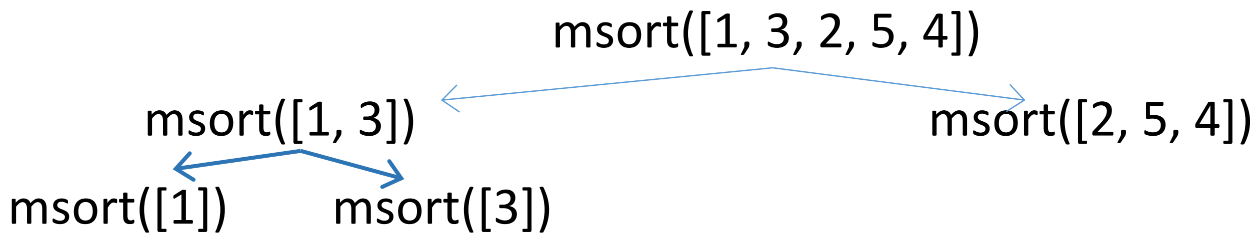
Mergesort: example

msort([1, 3, 2, 5, 4])

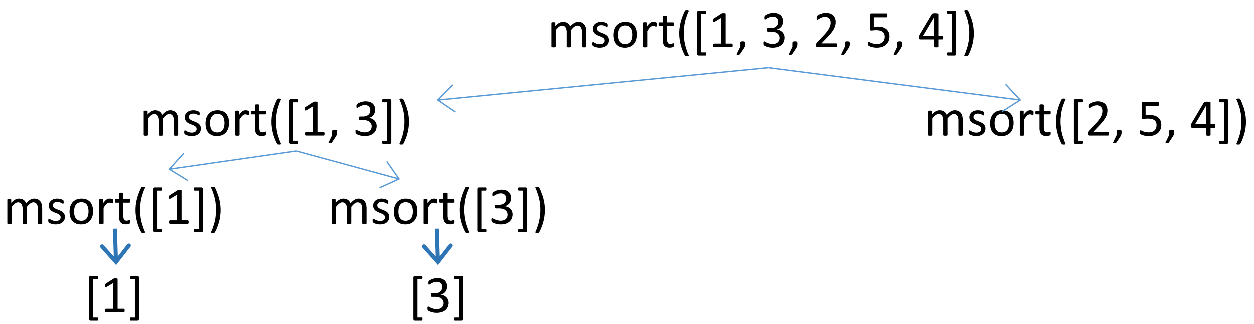
Mergesort: example



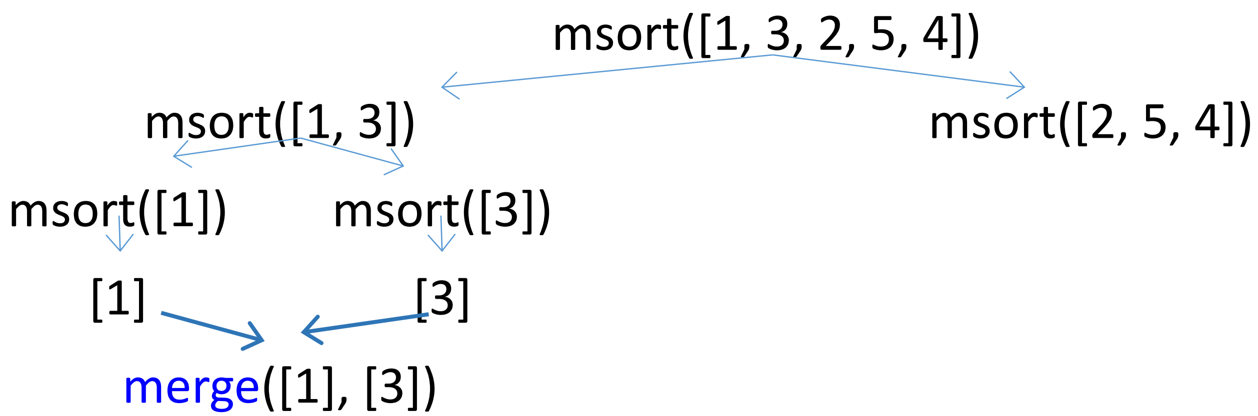
Mergesort: example



Mergesort: example



Mergesort: example



Mergesort: example

msort([1, 3, 2, 5, 4])

msort([1, 3]) msort([2, 5, 4])

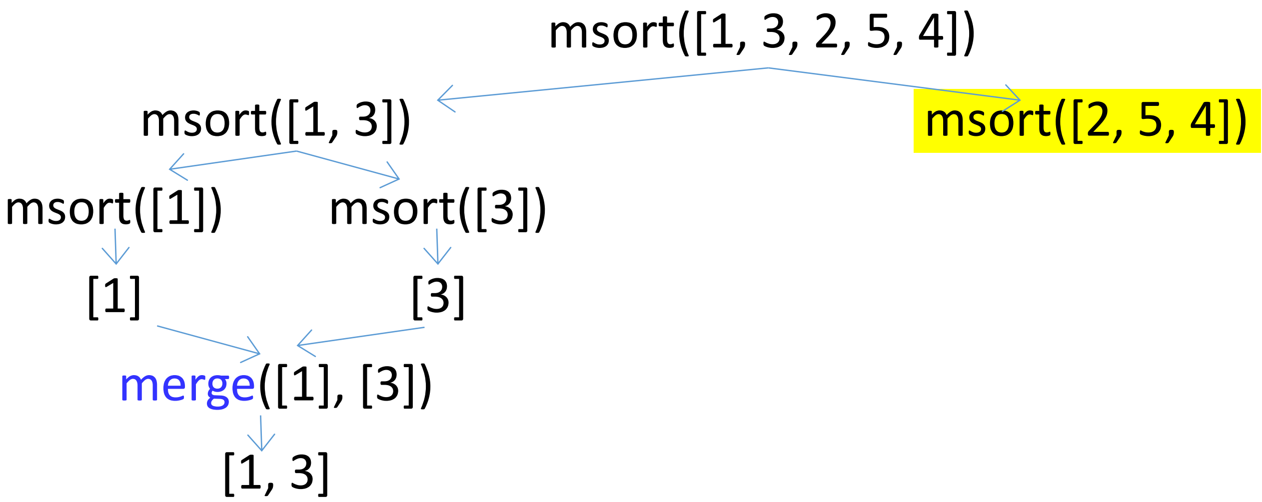
msort([1]) msort([3])

[1] [3]

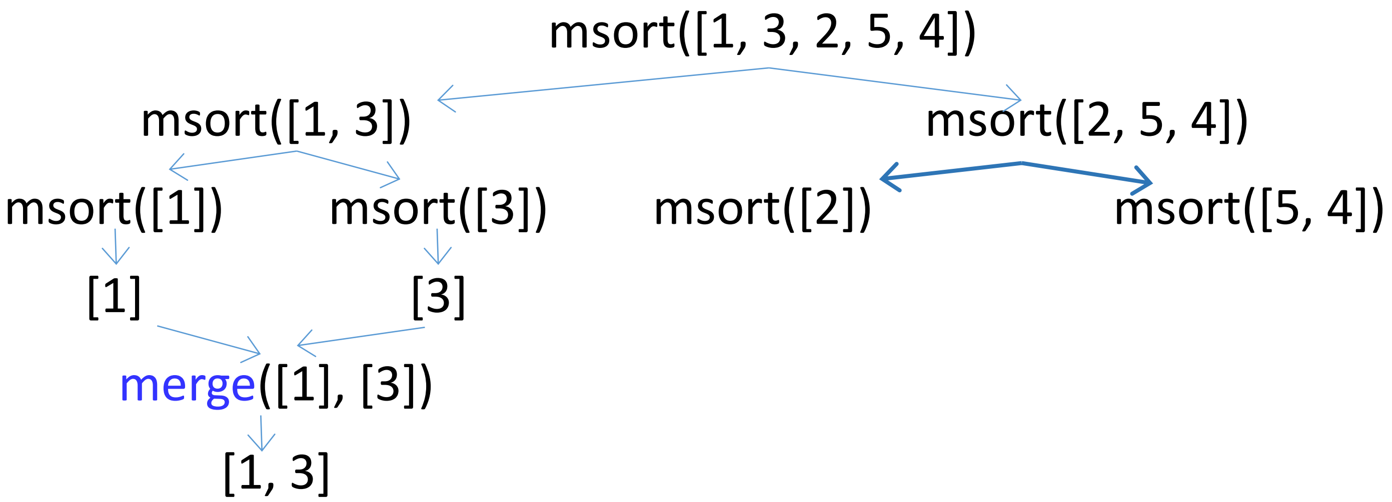
merge([1], [3])

[1, 3]

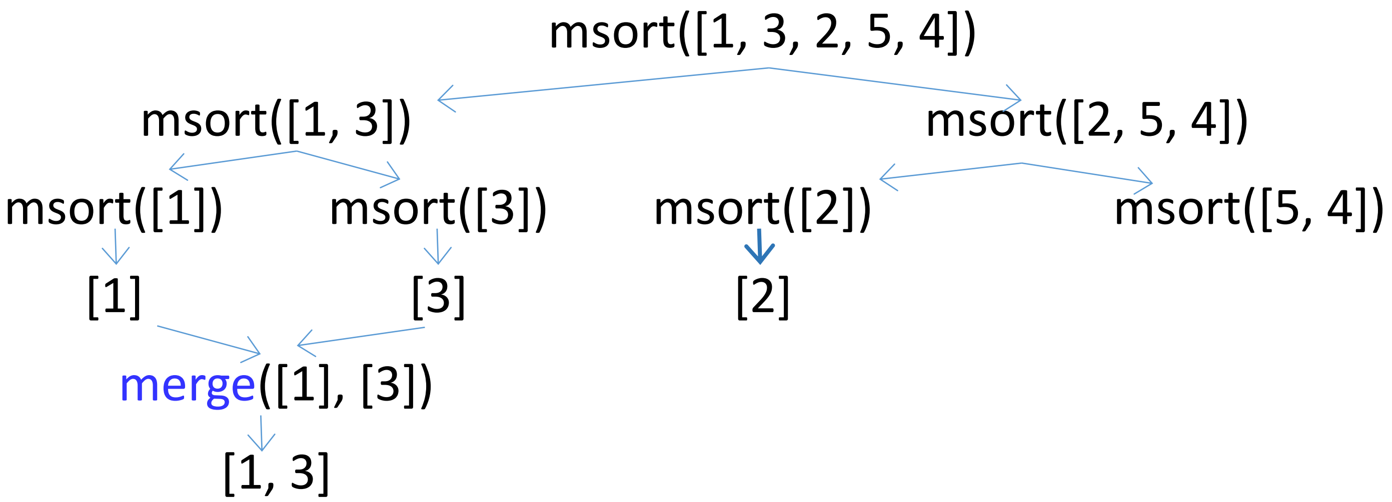
Mergesort: example



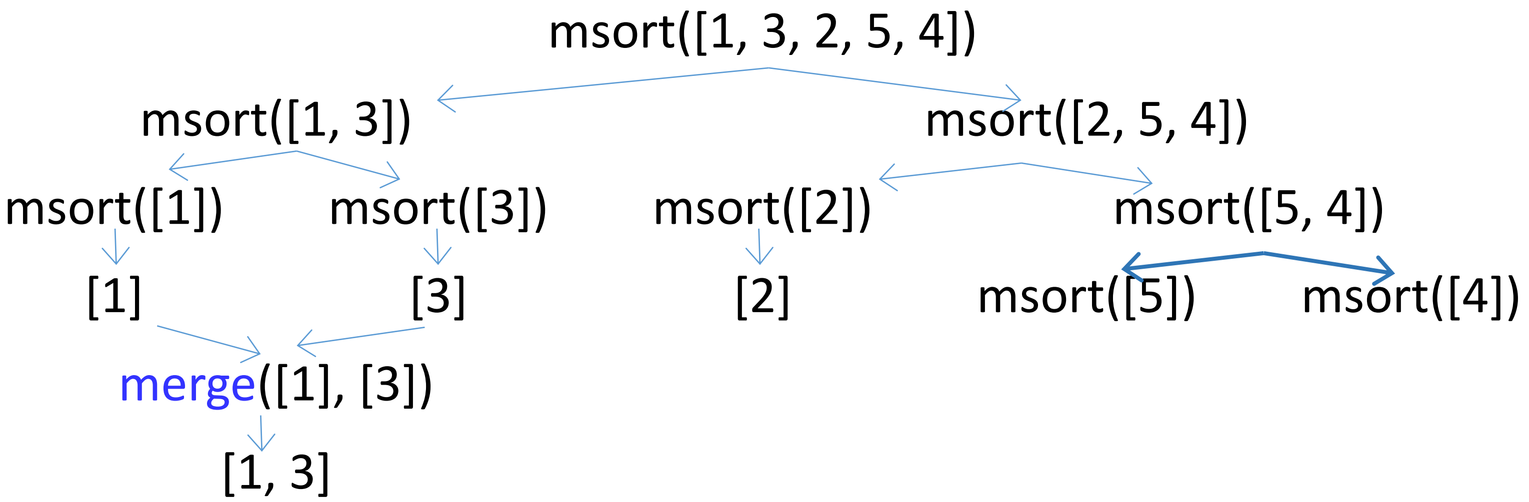
Mergesort: example



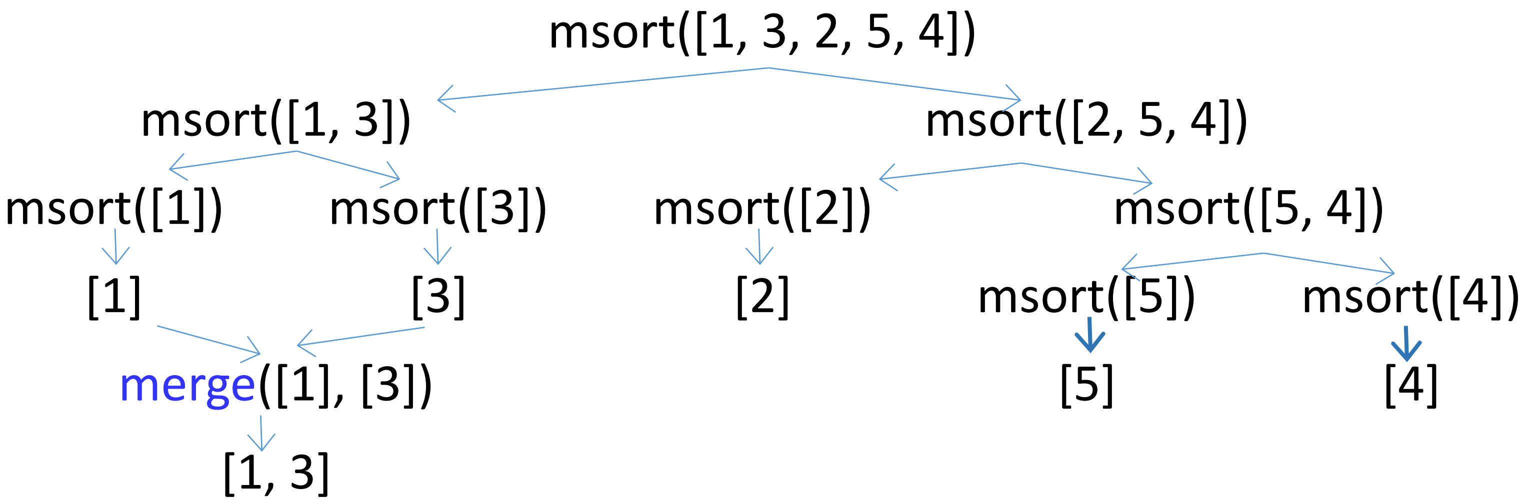
Mergesort: example



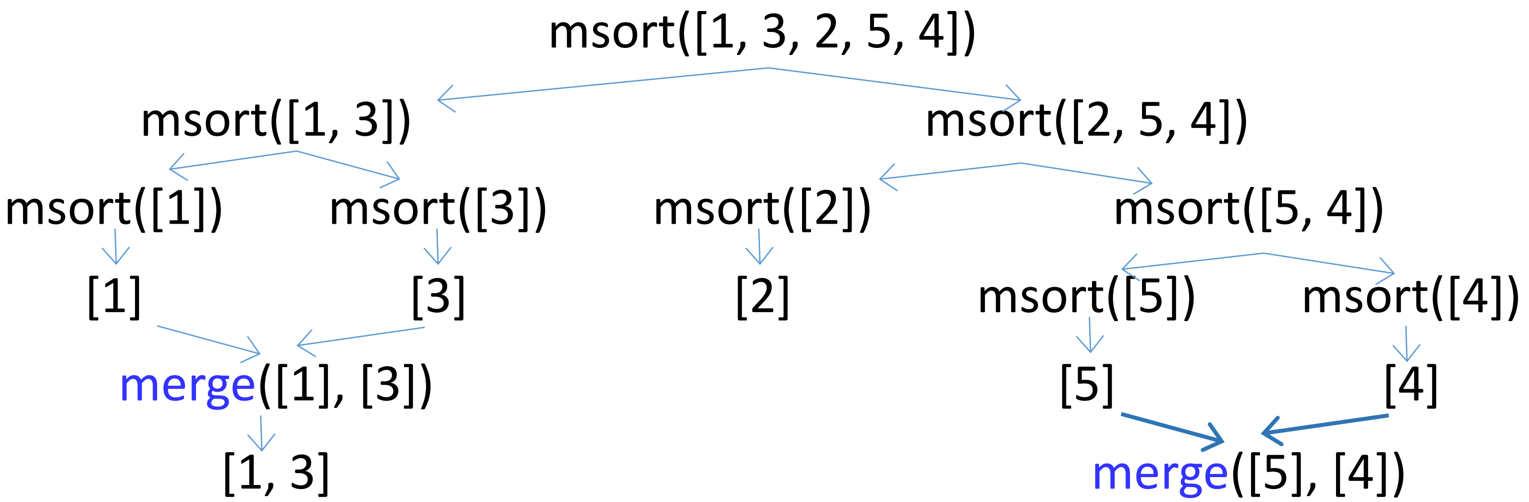
Mergesort: example



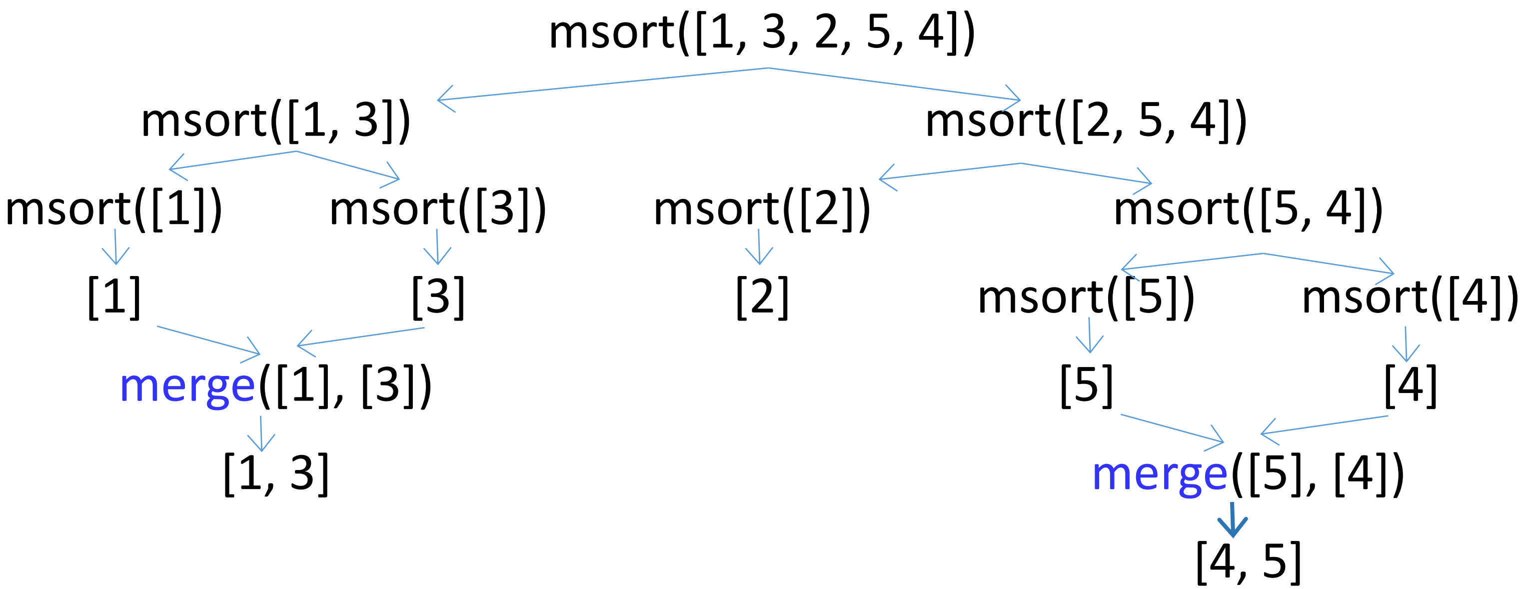
Mergesort: example



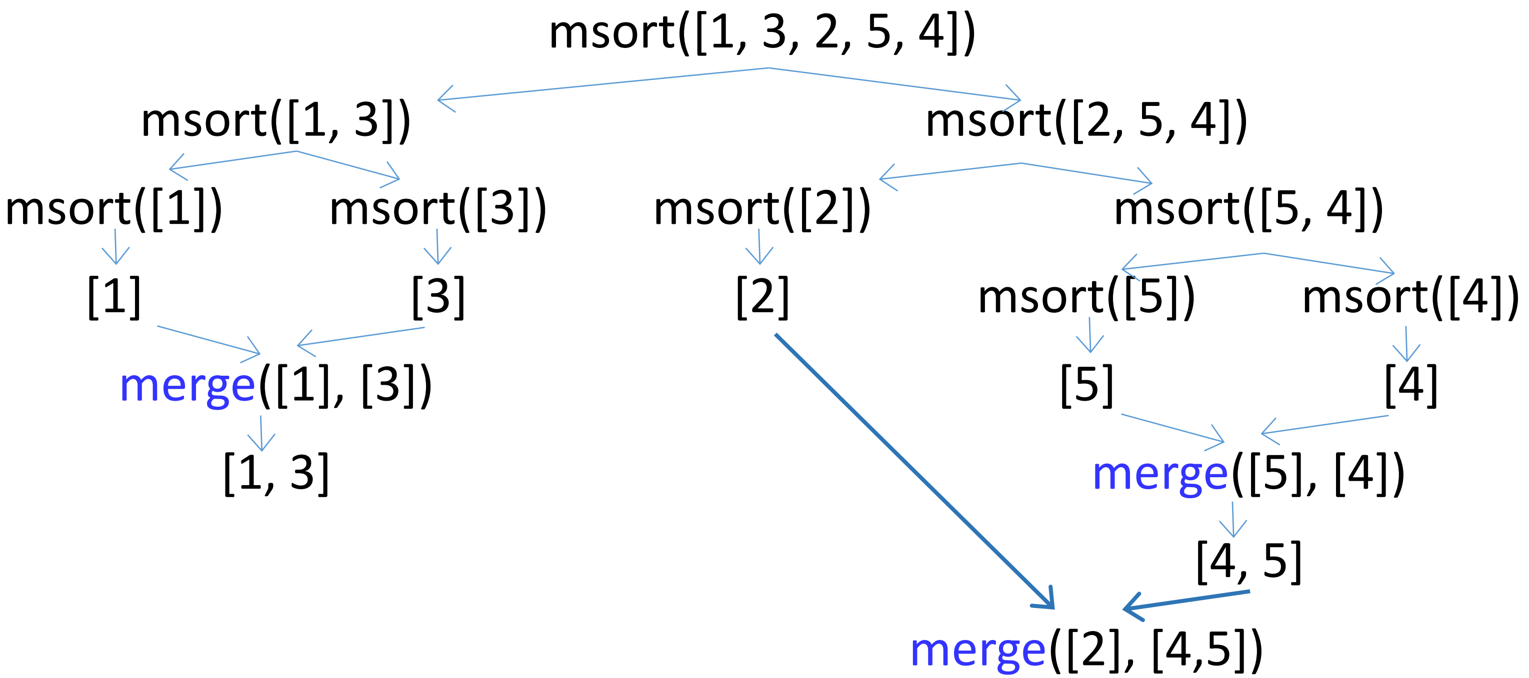
Mergesort: example



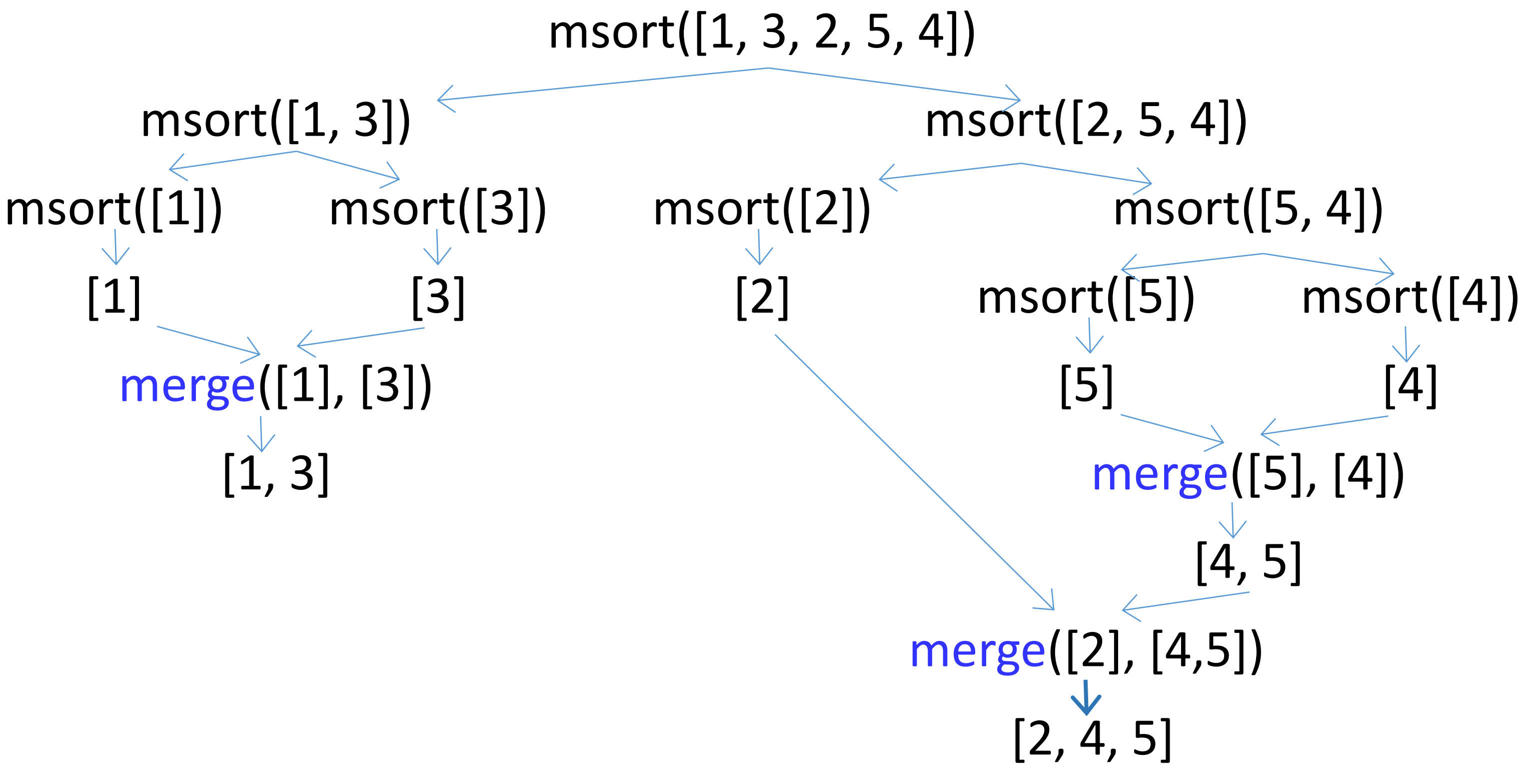
Mergesort: example



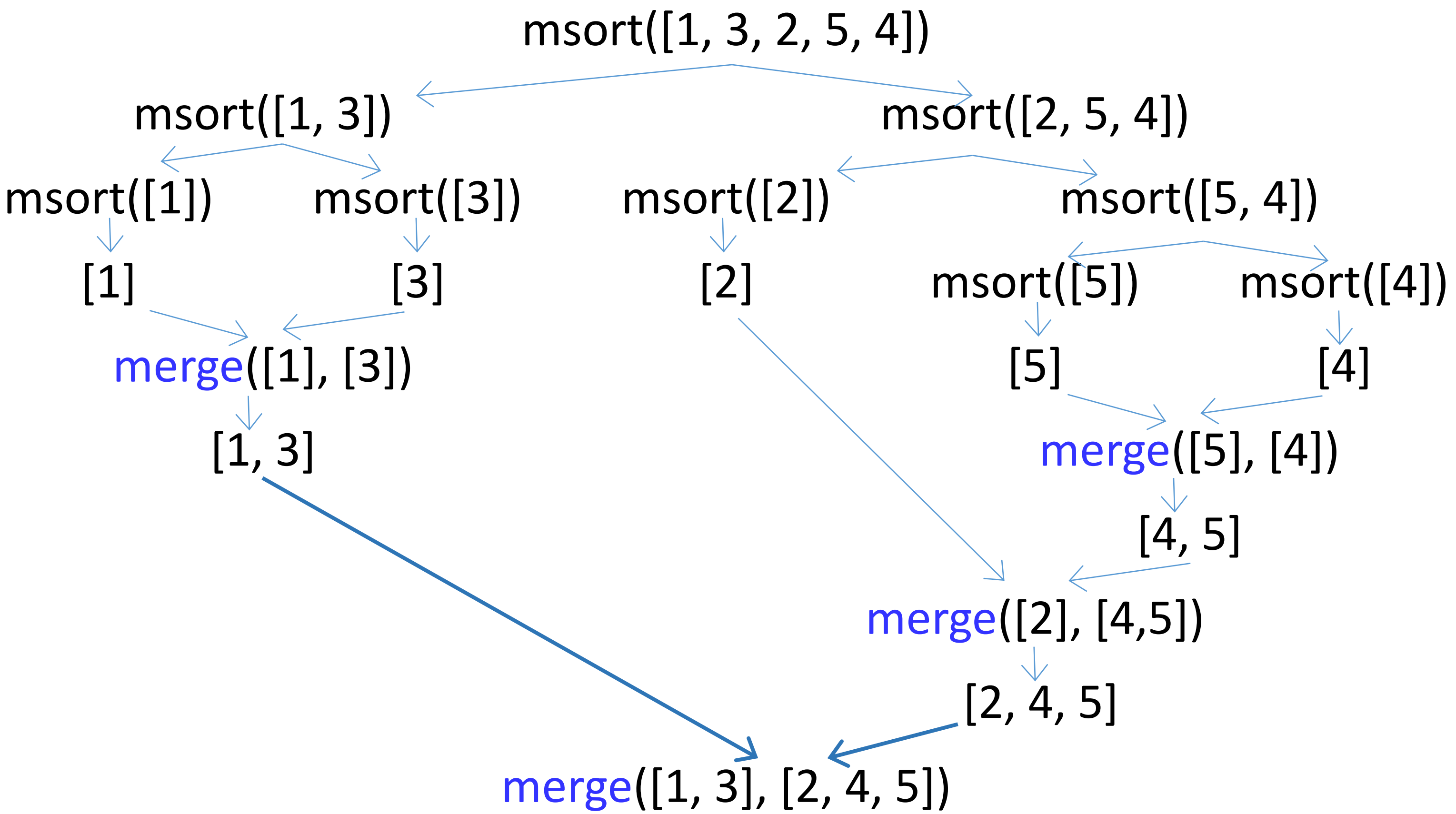
Mergesort: example



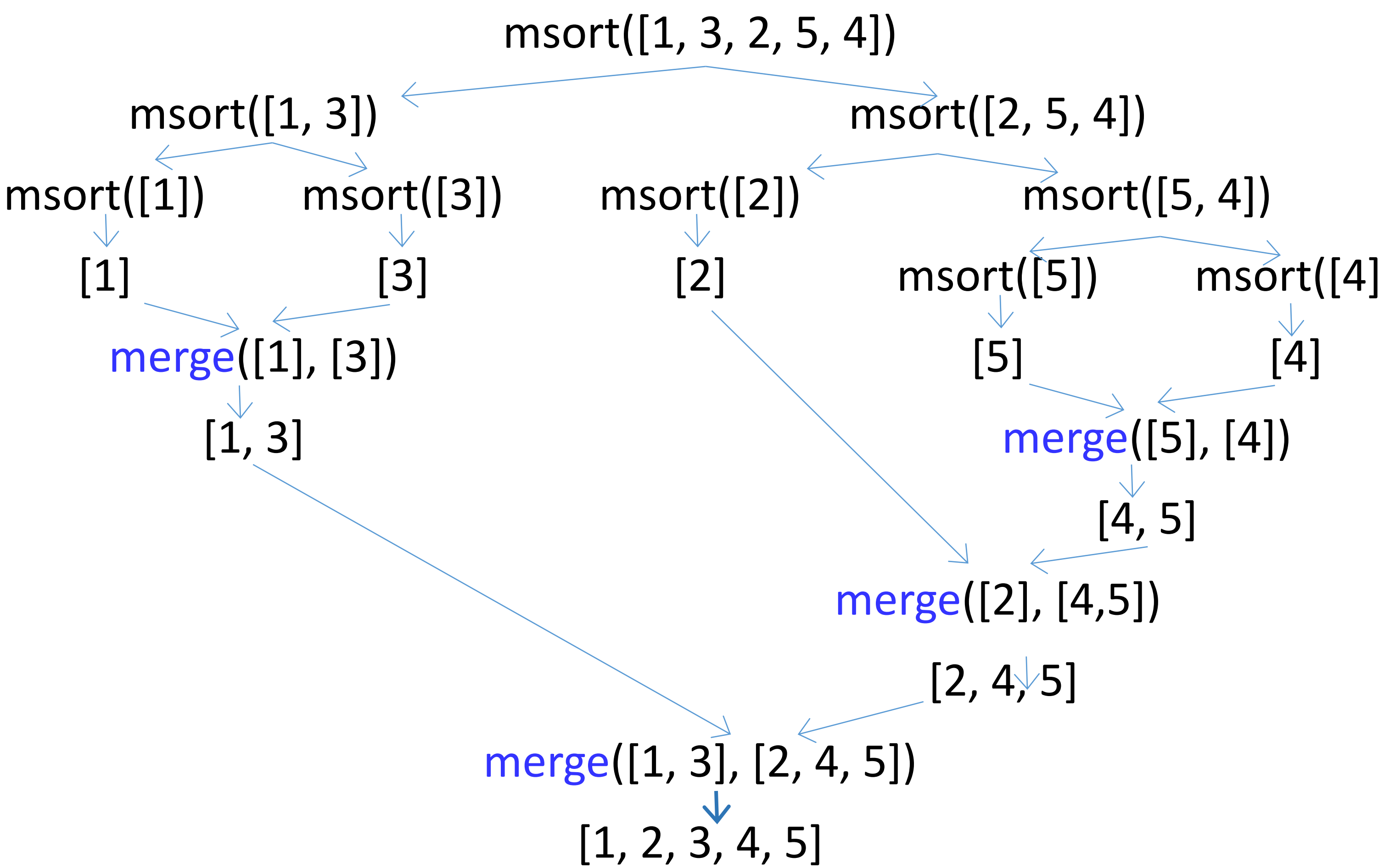
Mergesort: example



Mergesort: example



Mergesort: example



recursion: summary

Recursion: summary

- Recursion offers a way to express repetitive computations cleanly and succinctly
- How to:
  - what are the values used in the recursive call?
  - base case: when does the recursion stop?
  - recursive case:
    - what does a single round of computation involve?
    - what is the “smaller problem” to recurse on?
- Recursion is an essential component of every good computer scientist’s toolkit