CSc 120
Introduction to Computer Programming II

Adapted from slides by
Dr. Saumya Debray

13: Recursion

Volunteers!

- **Volunteers needed in front of the class for an activity:**
- Must be able to do simple addition

  (ex: 6 + 10 = 16)
- Must be able to speak

How much money is in this cup?

```
If the cup is not empty:
    Take out a coin. Pass the cup to the person on your
    left and ask them:
        "How much money is in this cup?"
    When they answer, tell the person on your right the
    sum of your coin and their answer
        (your_answer = your_coin + their_answer)

Else: # the cup is empty:
        Answer "zero" to the person on your right.
        (your_answer = 0)
```

Challenge

Can we express that procedure 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 cup[0] + how_much_money(cup[1:])
    else:
        return 0
```

5

[10, 1, 5]

Usage:

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

Calls and returns

```
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

- The input over which computation occurs is divided into two cases:
  - *base case* :
    - *do some computation and return the result*
  - *recursive case* :
    - *perform computation that reduces the size of the problem or input*
    - *make a recursive call to do the remainder of the computation*
- *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 cup[0] + how_much_money(cup[1:])
    else:
        return 0
```

recursive case

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 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 value

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 value

Problem 3

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

Problem 4

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 value

EXERCISE

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([10, 20 ,30], "--")
'10--20--30'
```

Recursion

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 to stop the repetition
- *Recursive case(s)*
  - what is the "rest of the computation" - i.e., the *smaller problem* to pass to the recursive call
- Note: the recursive case can be written in many ways. Revisit summing a list.

Versions of sumlist

Version 1	

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

current position in L is simply the head of L

adds current element of L

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[:n-1]) + L[n-1]
```

current position in L is the last element of L

argument to recursive call is "rest of the list" up to the last element (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[:n-1]) + L[n-1]
```

Version 3
("smaller" need not be by just 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: ])
```

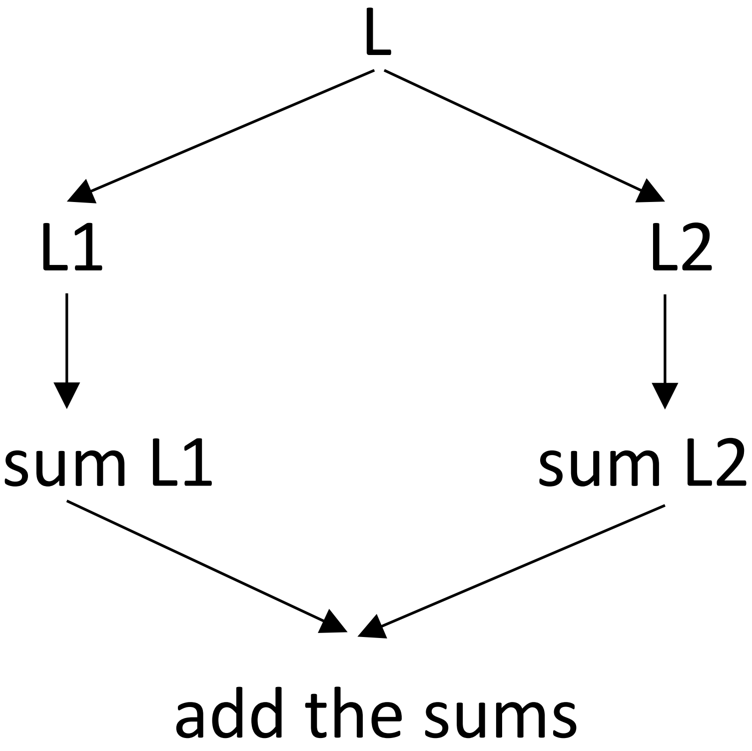
recursive sumlist

input list

split into two halves

add the halves (recursively)

return the sum of the sums



L

L1

L2

sum L1

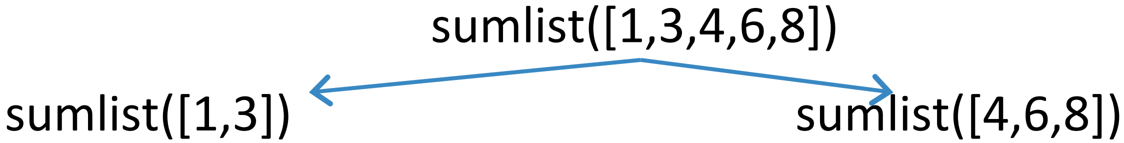
sum L2

add the sums

sumlist: example

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

sumlist: example

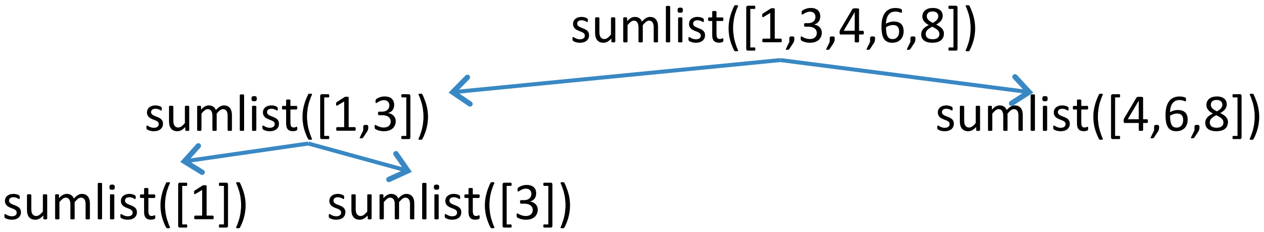


sumlist([1,3,4,6,8])

sumlist([1,3])

sumlist([4,6,8])

sumlist: example



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

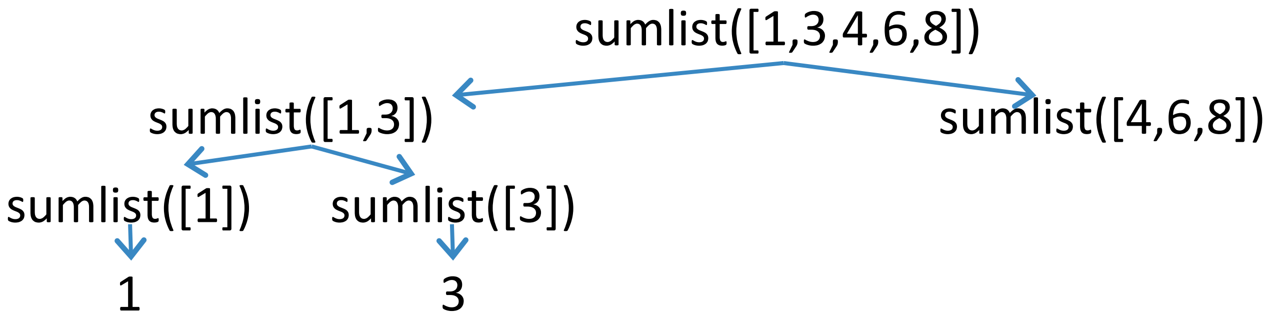
sumlist([1,3])

sumlist([4,6,8])

sumlist([1])

sumlist([3])

sumlist: example



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

sumlist([1,3])

sumlist([4,6,8])

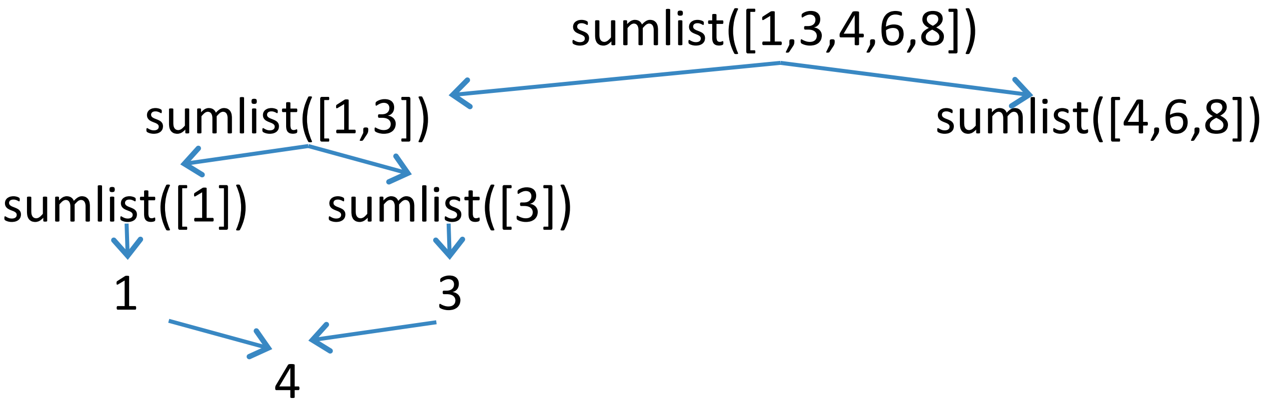
sumlist([1])

sumlist([3])

1

3

sumlist: example



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

sumlist([1,3])

sumlist([4,6,8])

sumlist([1])

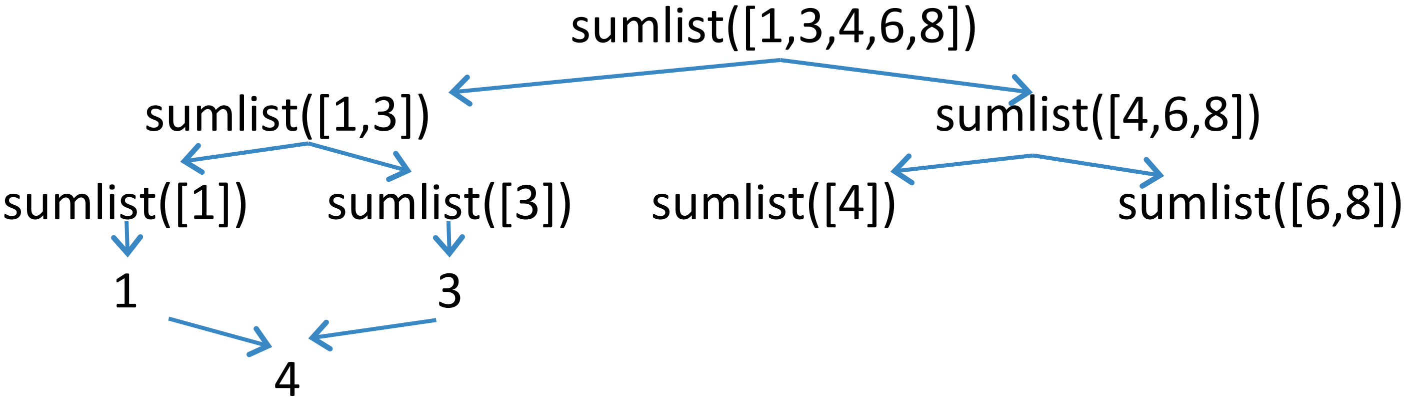
sumlist([3])

1

3

4

sumlist: example



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

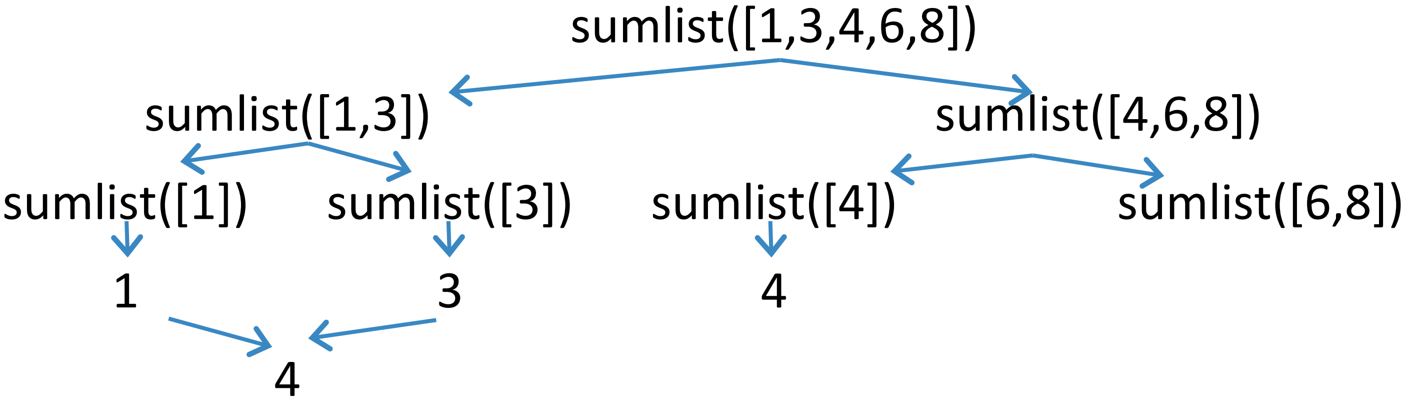
sumlist([1,3]) sumlist([4,6,8])

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

1 3

4

sumlist: example



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

sumlist([1,3])

sumlist([4,6,8])

sumlist([1])

sumlist([3])

sumlist([4])

sumlist([6,8])

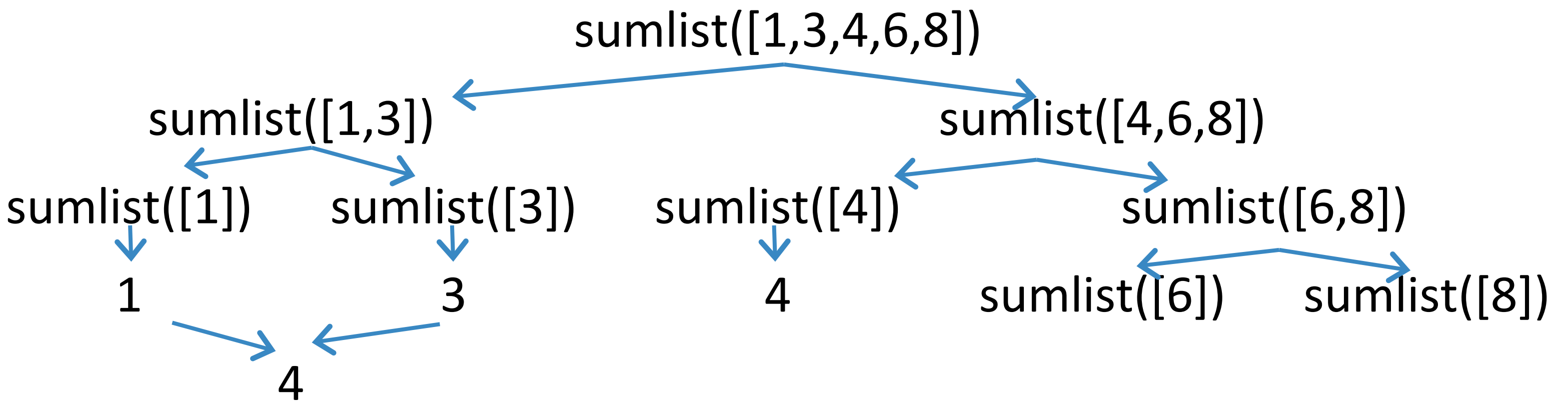
1

3

4

4

sumlist: example



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

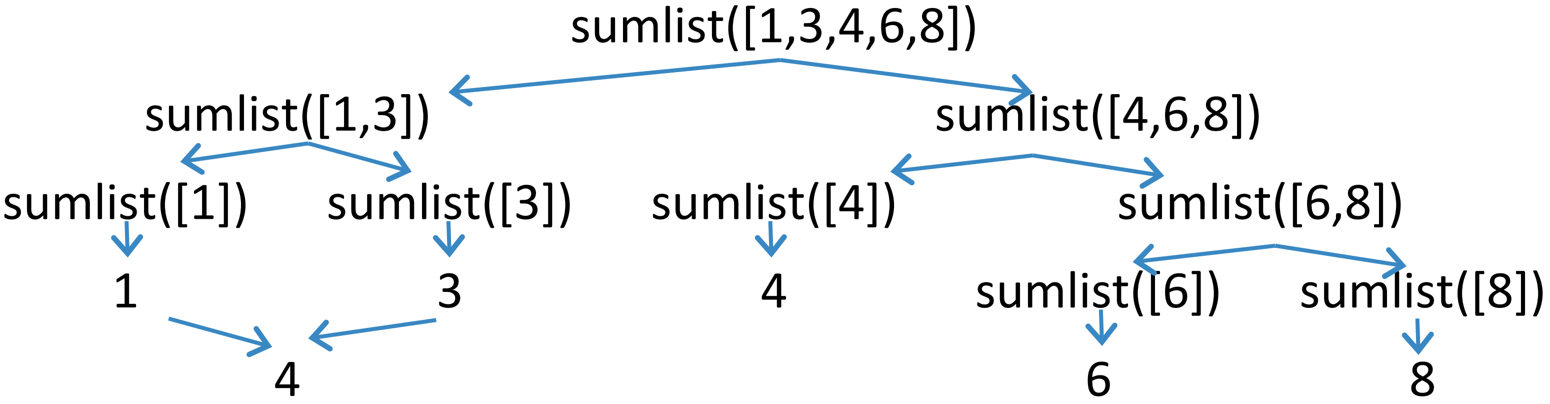
sumlist([1,3]) sumlist([4,6,8])

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

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

4

sumlist: example



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

sumlist([1,3])

sumlist([4,6,8])

sumlist([1])

sumlist([3])

sumlist([4])

sumlist([6,8])

1

3

4

sumlist([6])

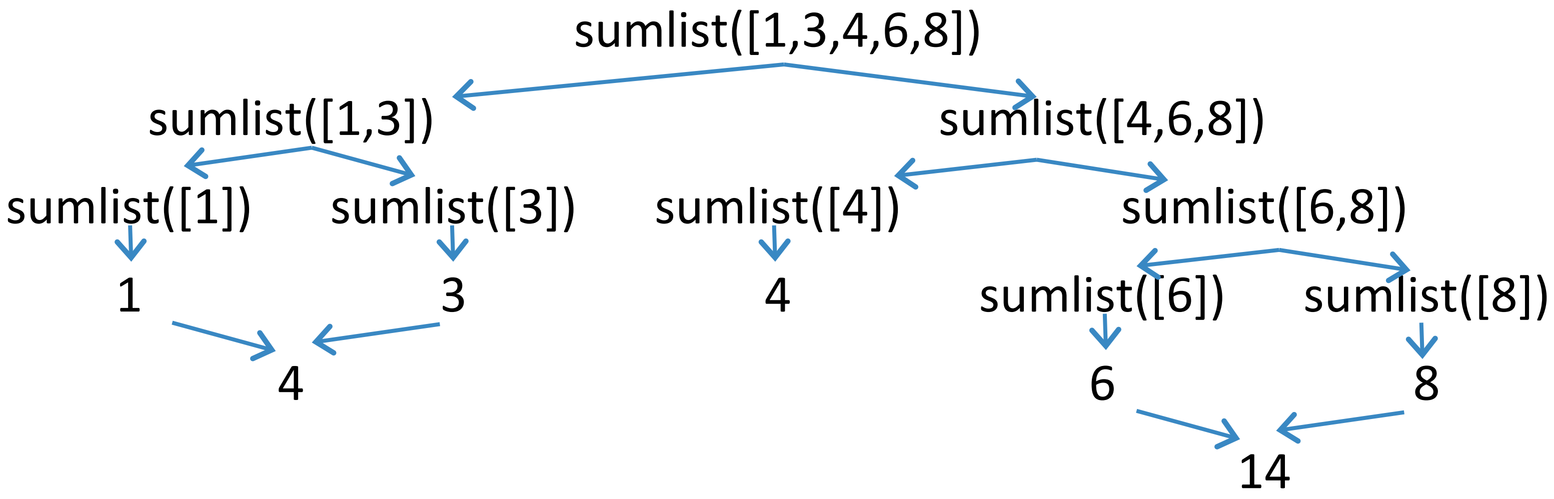
sumlist([8])

4

6

8

sumlist: example



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

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

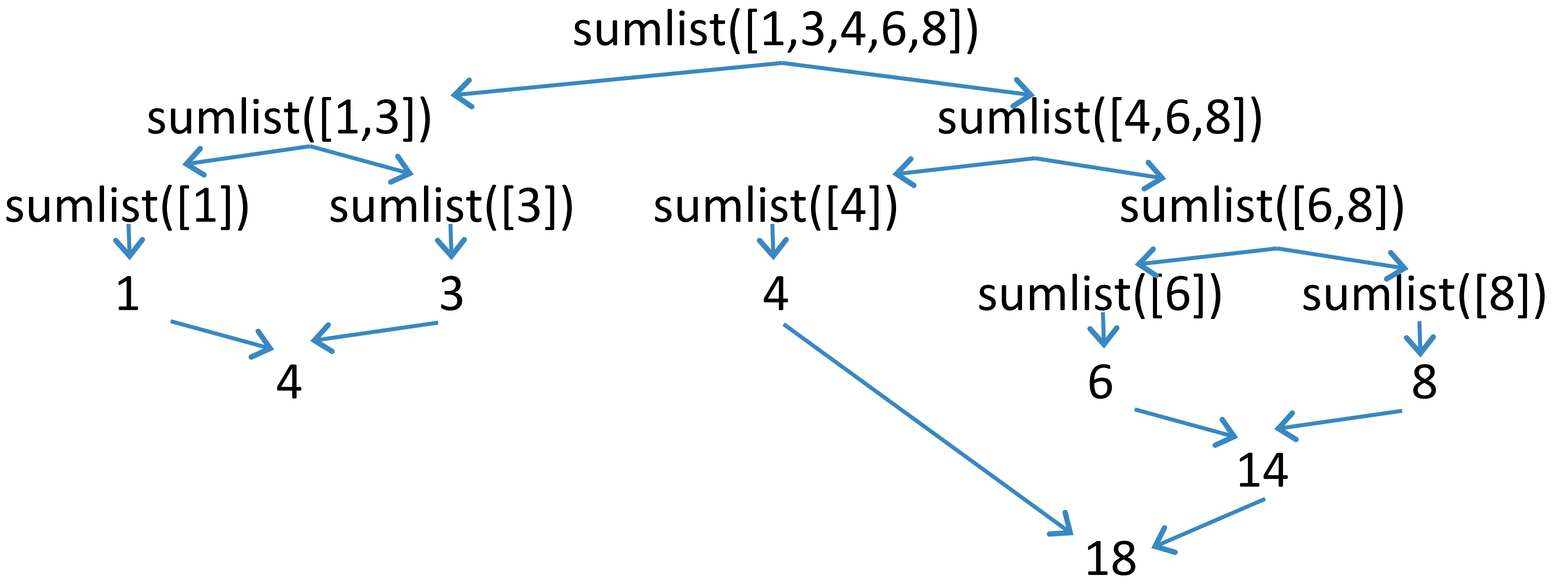
sumlist([1]) sumlist([3]) sumlist([4]) sumlist([6,8])

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

4 6 8

14

sumlist: example



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

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

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

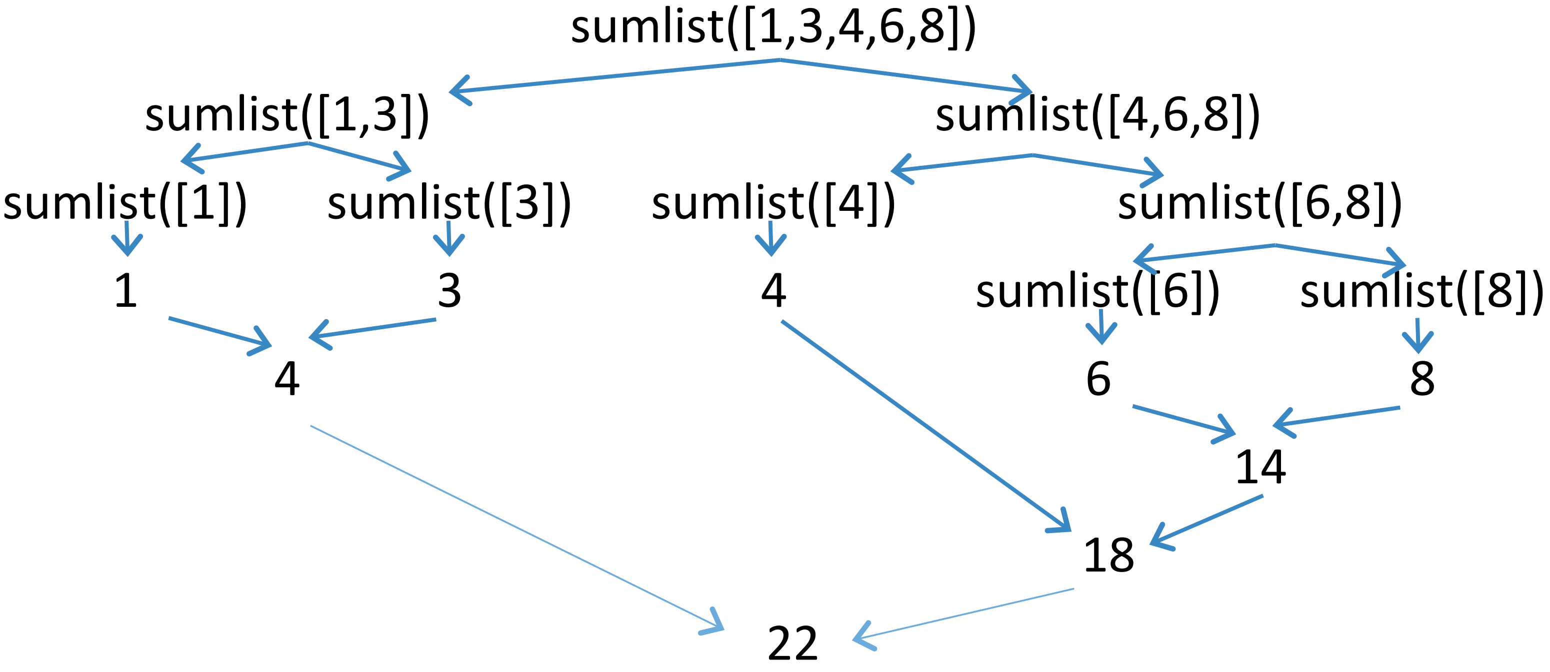
1 3 4 sumlist([6]) sumlist([8])

4 6 8

14

18

sumlist: example



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

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

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

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

4 6 8

14

18

22

Recursion: how to

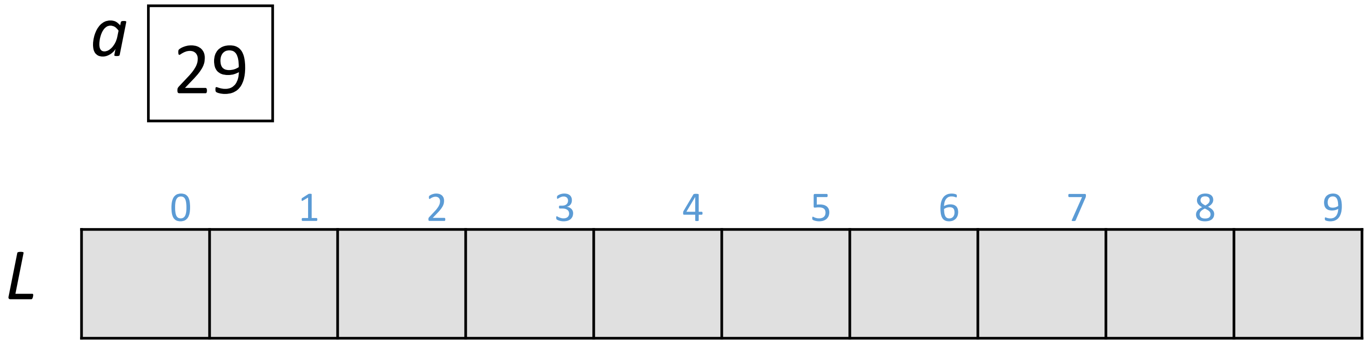
To write a recursive function, figure out:

- *What values are involved in the computation?*
  - these will be the arguments to the recursive function
- *Base case(s)*
  - when to stop the repetition
- *Recursive case(s)*
  - what is the "rest of the computation" - i.e., the *smaller problem* to pass to the recursive call

recursion: example
binary search

Searching a sorted list

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

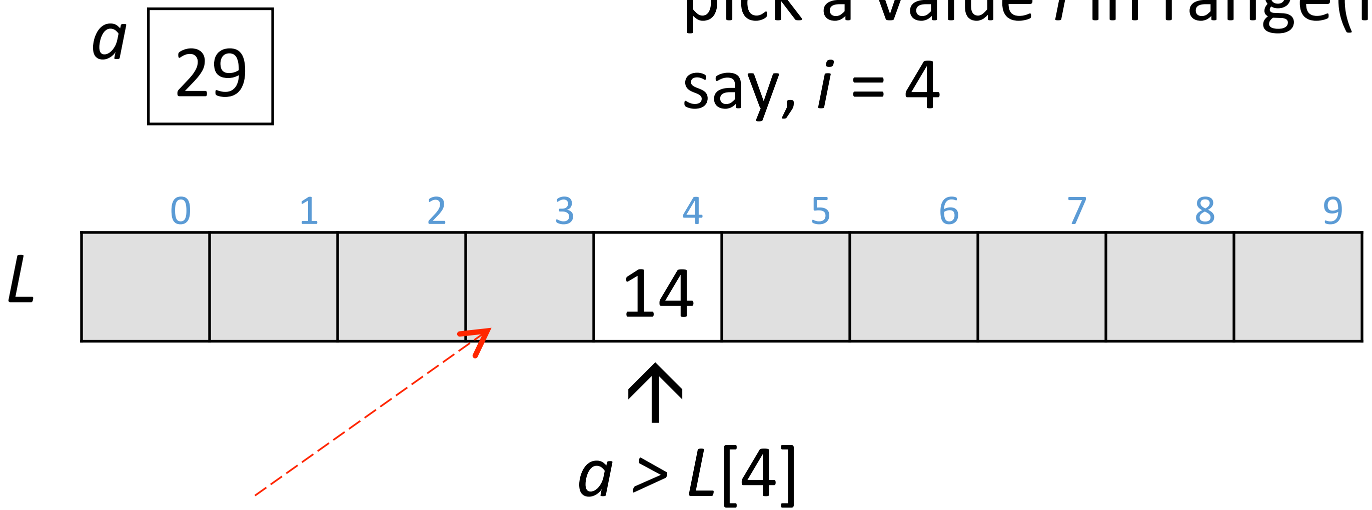


Searching a sorted list

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

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

say, *i* = 4



a > *L*[4]

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

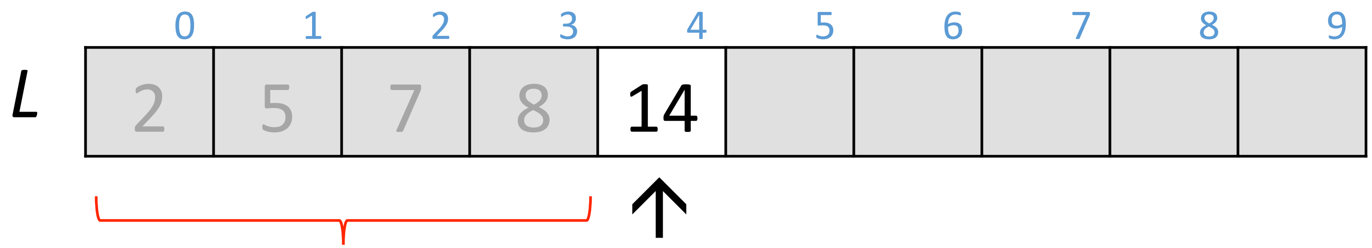
Searching a sorted list

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

a 29

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

```
bin_search(list, item)
    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 – write the code

Binary search: complexity

- The size of the search area is halved at each round of repetition

Comparisons	Approx. number of items left
1	n/2
2	n/4
3	n/8
…	…
i	$n/2^i$

  - The number of comparisons until we are done is

    i, where $n/2^i = 1$

    solving for i gives i = log n
  - total no. of rounds of repetition (recursion) = $\log_2(n)$

Binary search: complexity

- The size of the search area is halved at each round of repetition (recursion)
  - total no. of rounds of repetition = $\log_2(n)$

    or the number of comparisons is $log_2(n)$
- However, on each round of repetition, the work done is ***not*** a fixed amount due to slicing
  - slicing is O(n)
- Fix that by computing the indices and passing them as parameters.

Binary search: no slicing

```
def bin_search(L, item, lo, hi):
    if lo > hi:
        return False
    if lo == hi:
        return L[lo] == item

    mid = (lo+hi)//2
    if item <= L[mid]:
        return bin_search(L, item, lo, mid)
    else:
        return bin_search(L, item, mid+1, hi)
```

Binary search: complexity

- The size of the search area is halved at each round of repetition (recursion)
  - total no. of rounds of repetition = $\log_2(n)$
- On each recursive step, the work done is a fixed amount
  - O(1)

∴ Overall complexity: O(log n)

recursion: example

Example: merging two sorted lists

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

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 repetition: ???

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

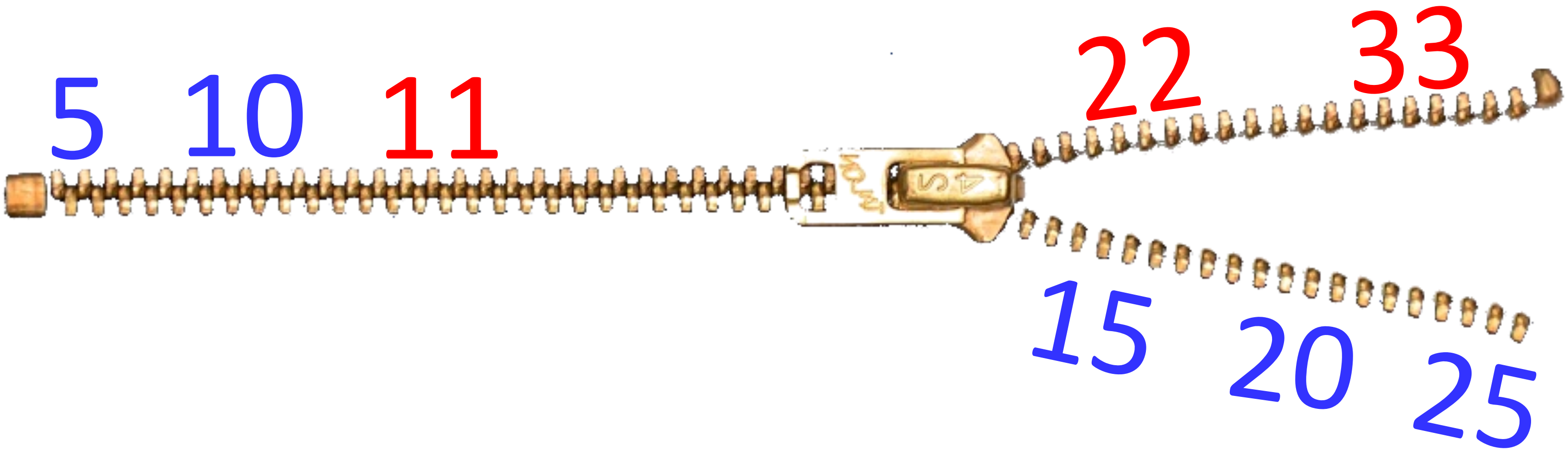
So the recursive function will look something like

```
def merge(L1, L2):
    …
```

Merging: repetition

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

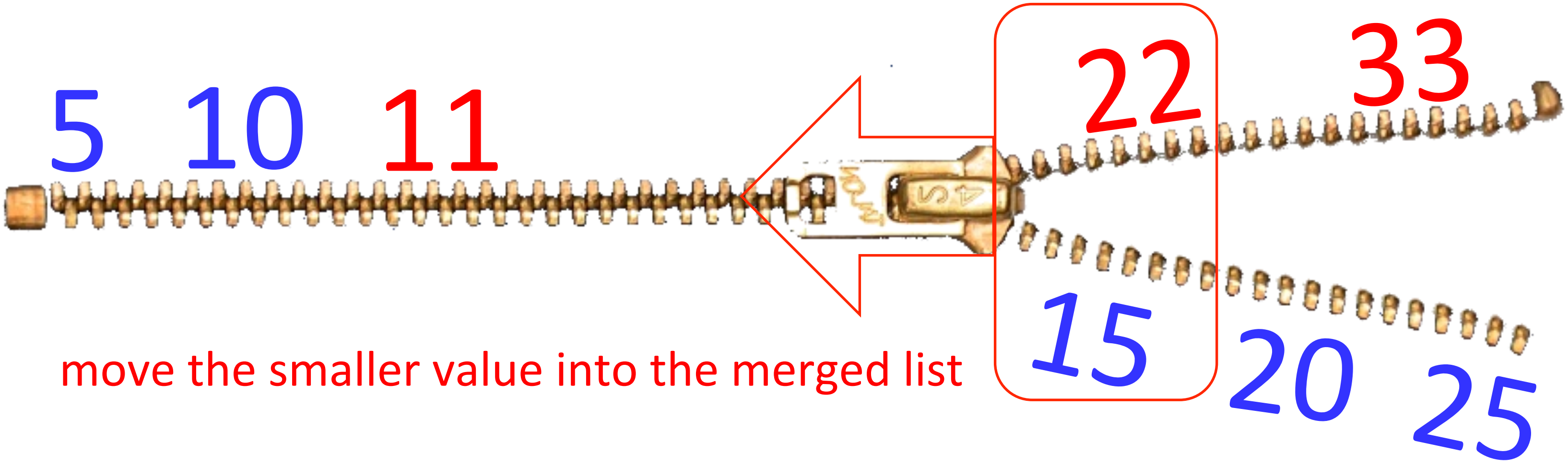
2. What does the computation involve in each call?



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?

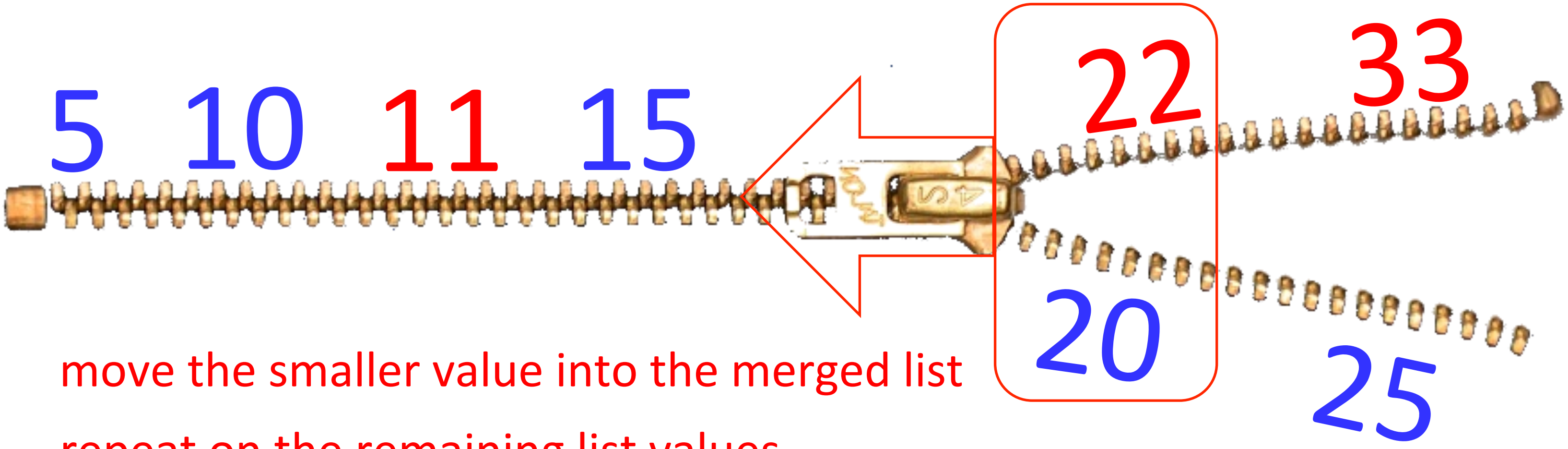


move the smaller value into the merged list

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?



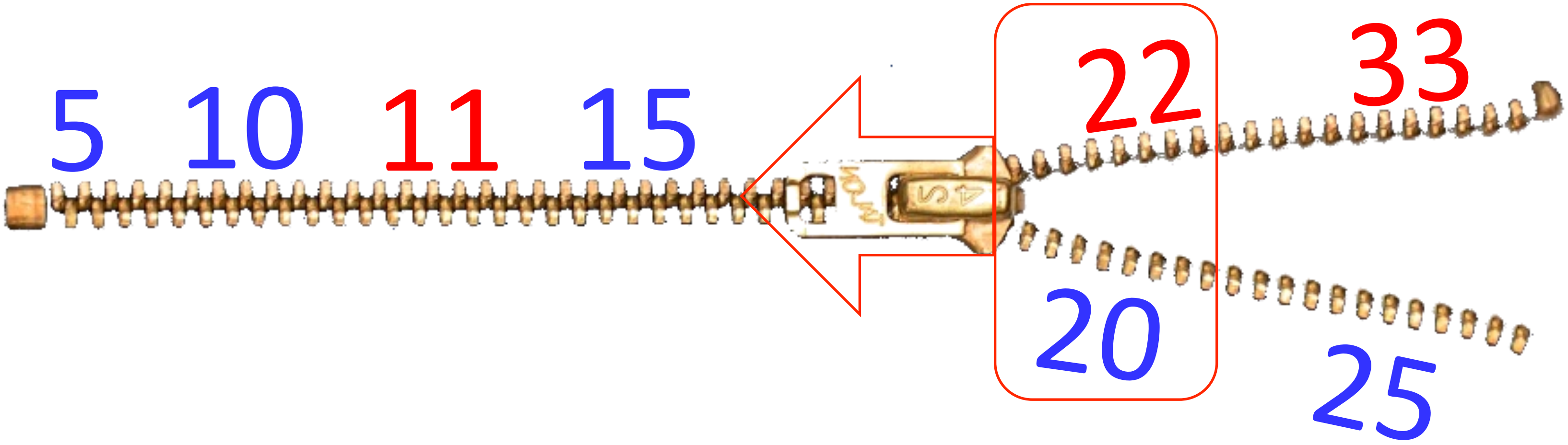
move the smaller value into the merged list

repeat on the remaining list values

Merging: base case

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

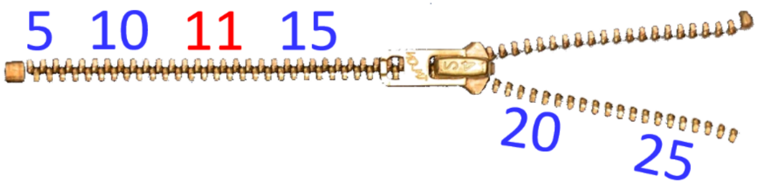
3. When can't we make the data smaller?



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):
        if L1 == []:
                return merged + L2
        elif L2 == []:
                return merged + L1
        else:
                ….
```

Merging: base case

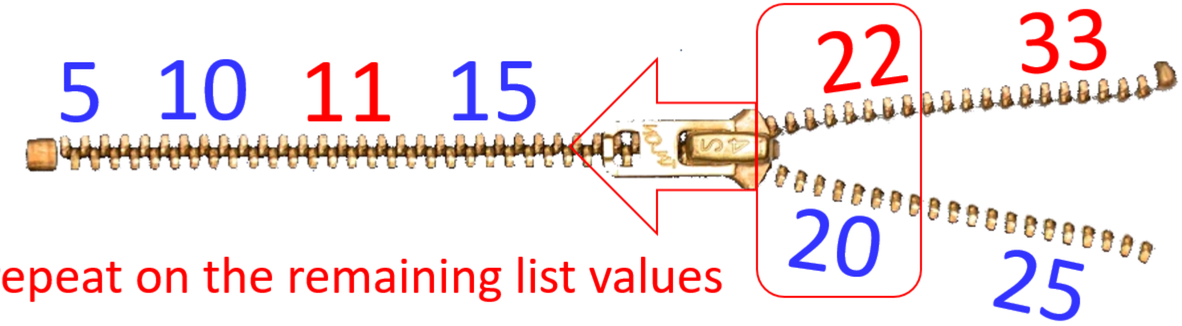
The code looks something like:

```
def  merge(L1, L2, merged):
        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"



repeat on the remaining list values

Merging: recursive case

```
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

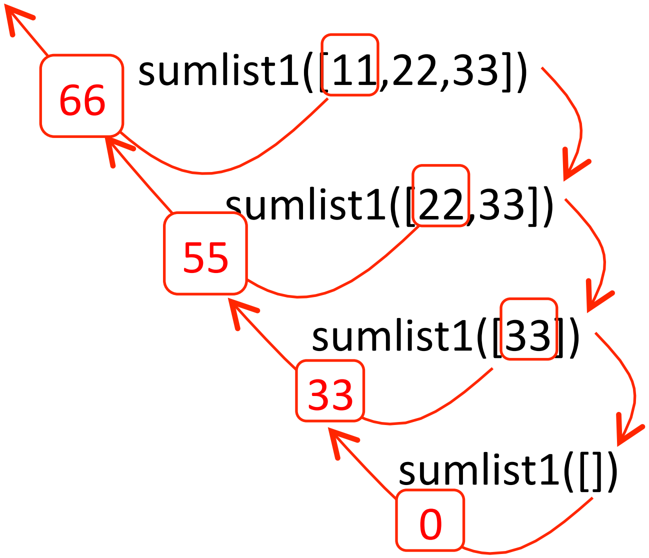
```
>>> def merge(L1,L2,merged):
        if L1 == [] or L2 == []:
                return merged + L1 + L2
        else:
                if L1[0] < L2[0]:
                        new_merged = merged + [L1[0]]
                        new_L1 = L1[1:]
                        new_L2 = L2
                else: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

Version 1

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



sumlist1([11,22,33])

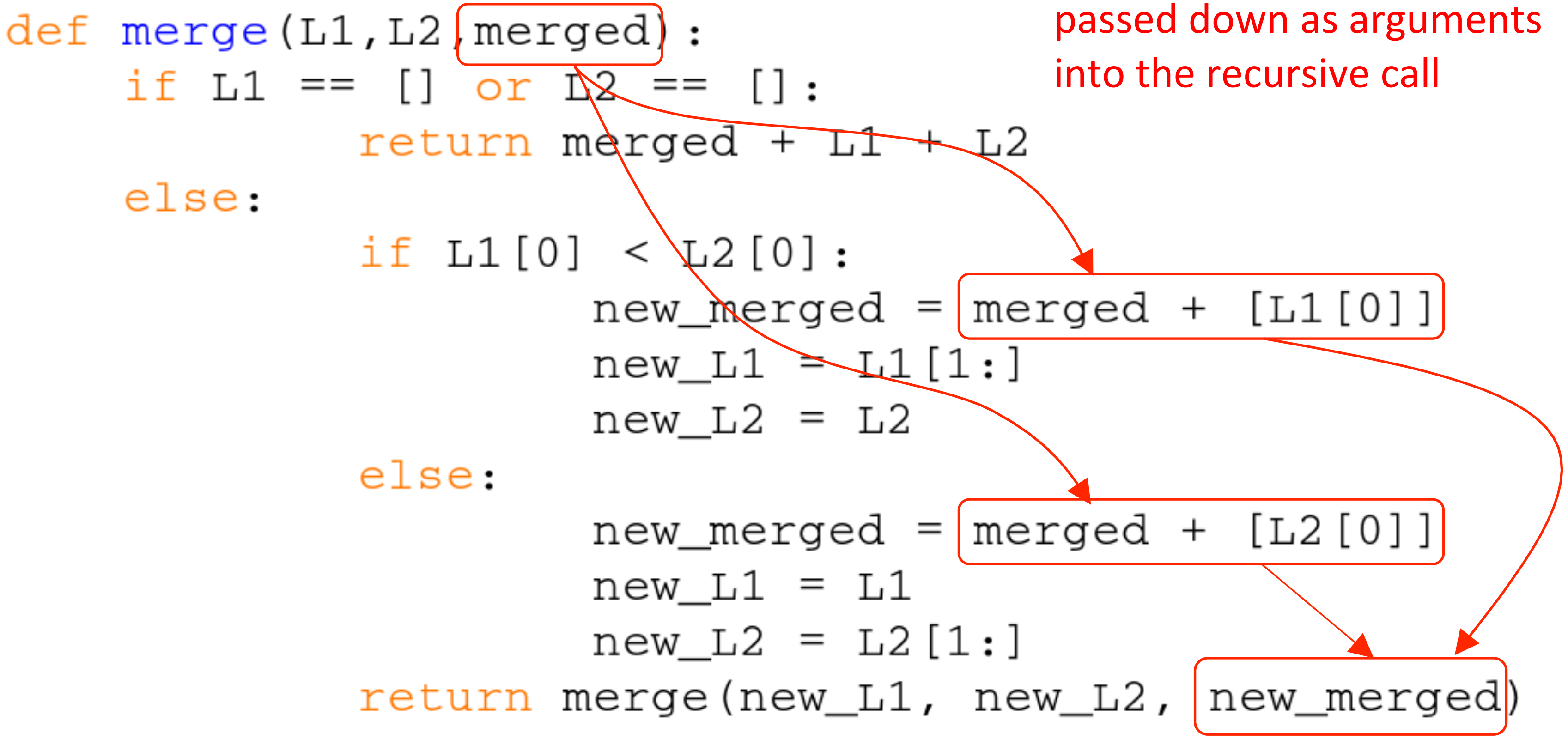
sumlist1([22,33])

sumlist1([33])

sumlist1([])

Recursion: flow of values

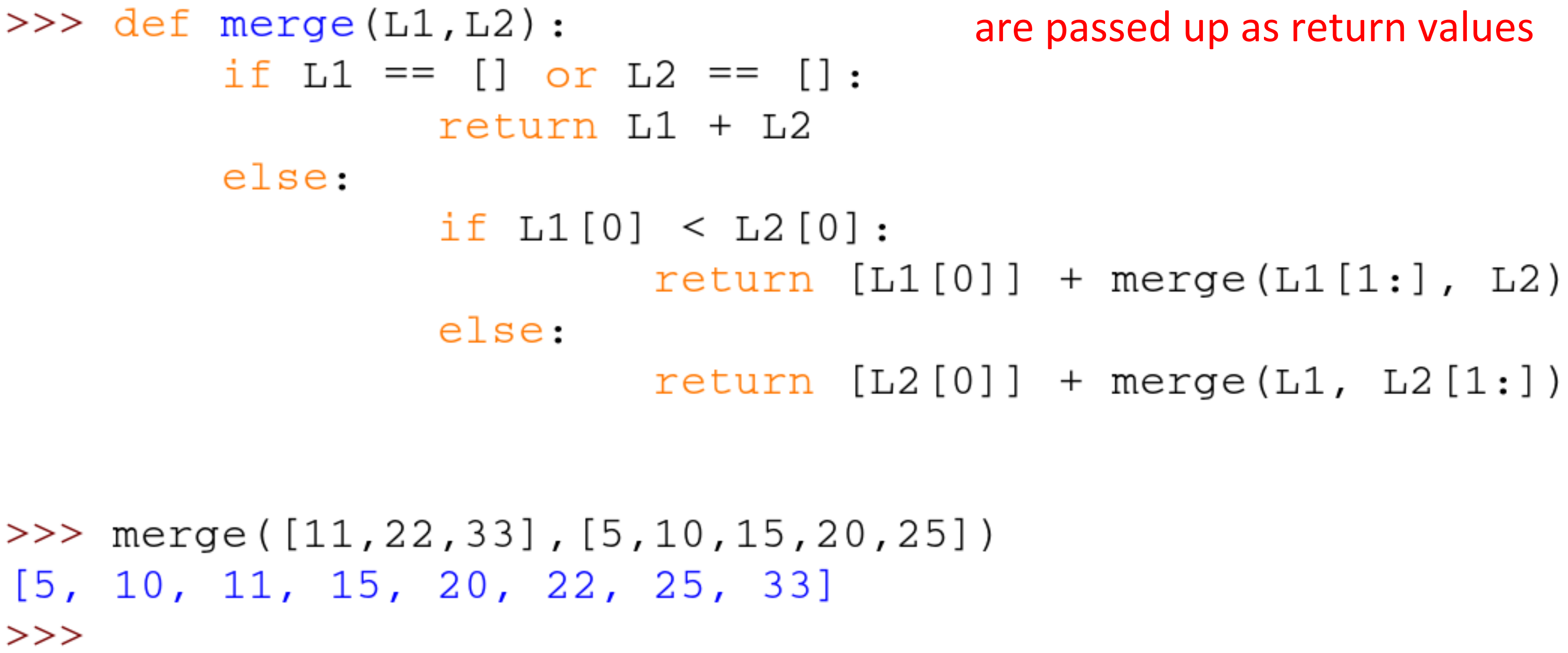
values are computed and passed down as arguments into the recursive call



```
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):
        if L1 == [] or L2 == []:
                return L1 + L2
        else:
                if L1[0] < L2[0]:
                        return [L1[0]] + merge(L1[1:], L2)
                else:
                        return [L2[0]] + merge(L1, L2[1:])


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

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

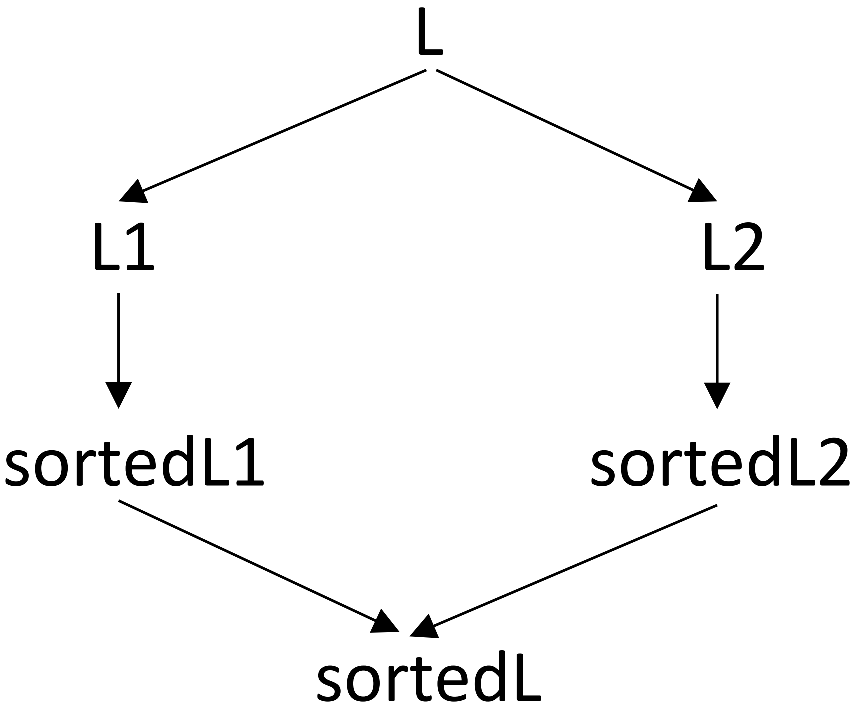
Algorithm: mergesort

input list

split into two halves

sort the halves recursively

merge the sorted lists



L

L1

L2

sortedL1

sortedL2

sortedL

Mergesort

- Base case: len(L) <= 1
  - no further halving possible

- Recursive case:
  - setting up the next round of computation: splitting 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

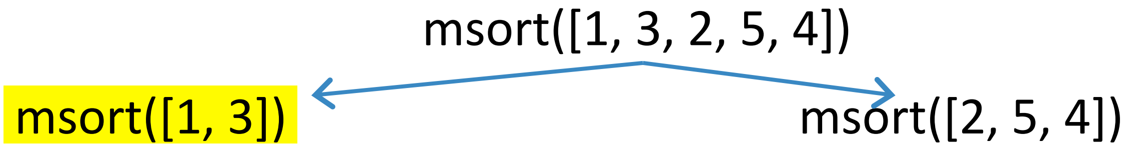
Mergesort

```
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

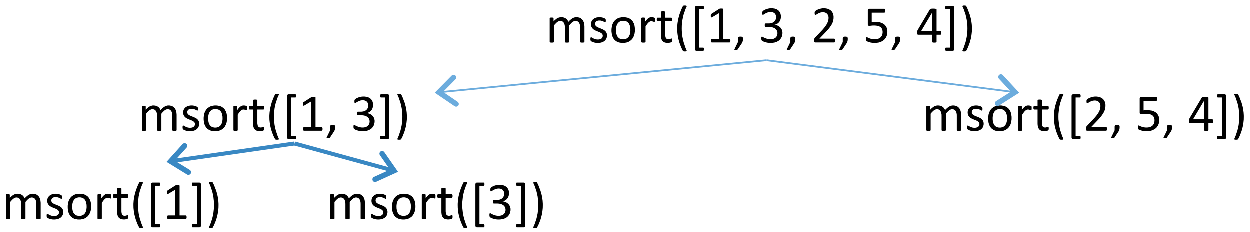


msort([1, 3, 2, 5, 4])

msort([1, 3])

msort([2, 5, 4])

Mergesort: example



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

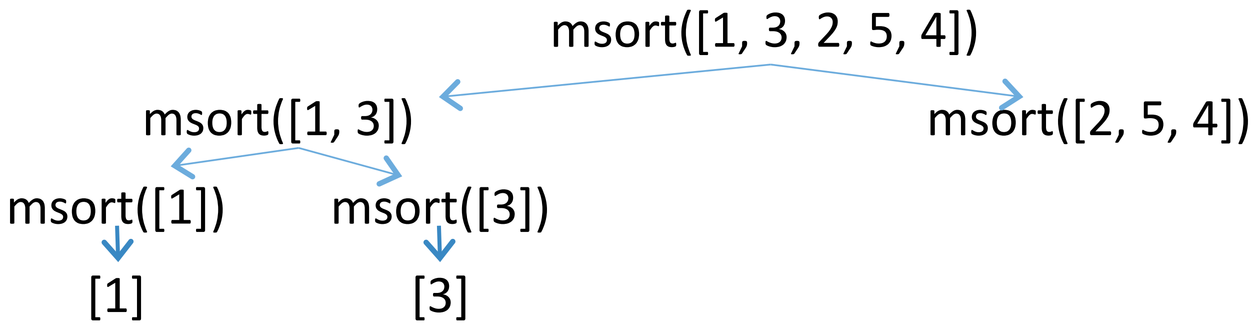
msort([1, 3])

msort([2, 5, 4])

msort([1])

msort([3])

Mergesort: example



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

msort([1, 3])

msort([2, 5, 4])

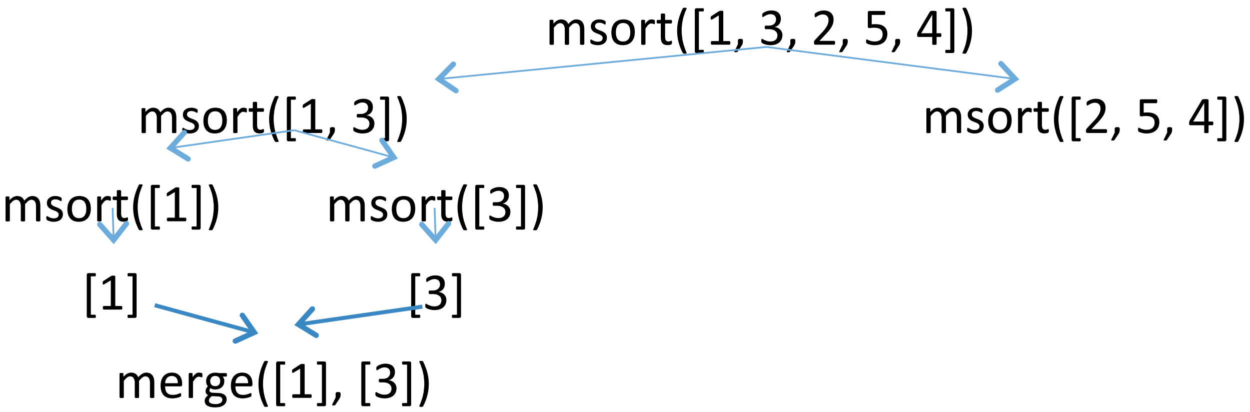
msort([1])

msort([3])

[1]

[3]

Mergesort: example



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

msort([1, 3])

msort([2, 5, 4])

msort([1])

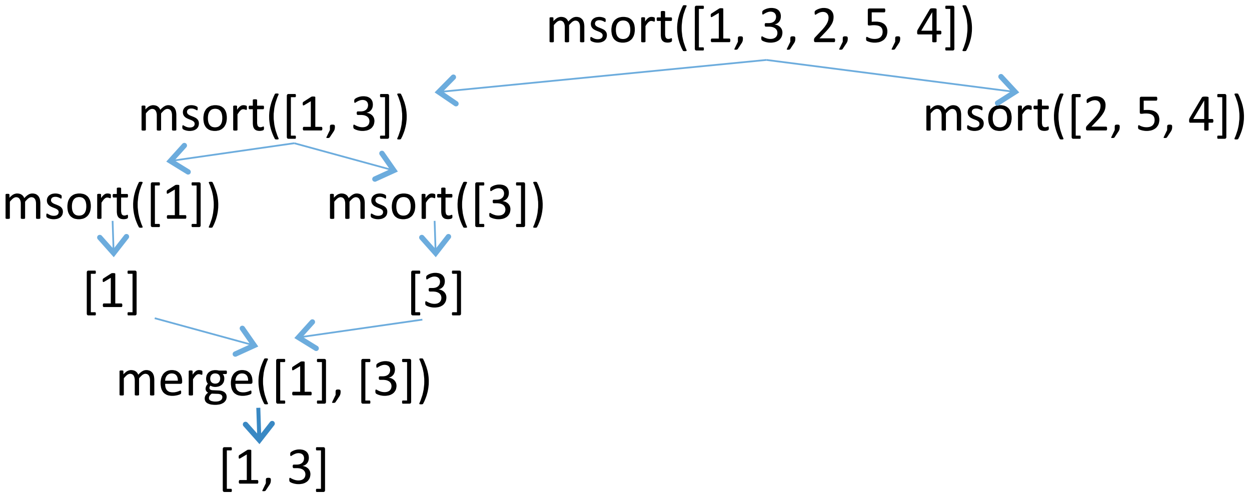
msort([3])

[1]

[3]

merge([1], [3])

Mergesort: example



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

msort([1, 3])

msort([2, 5, 4])

msort([1])

msort([3])

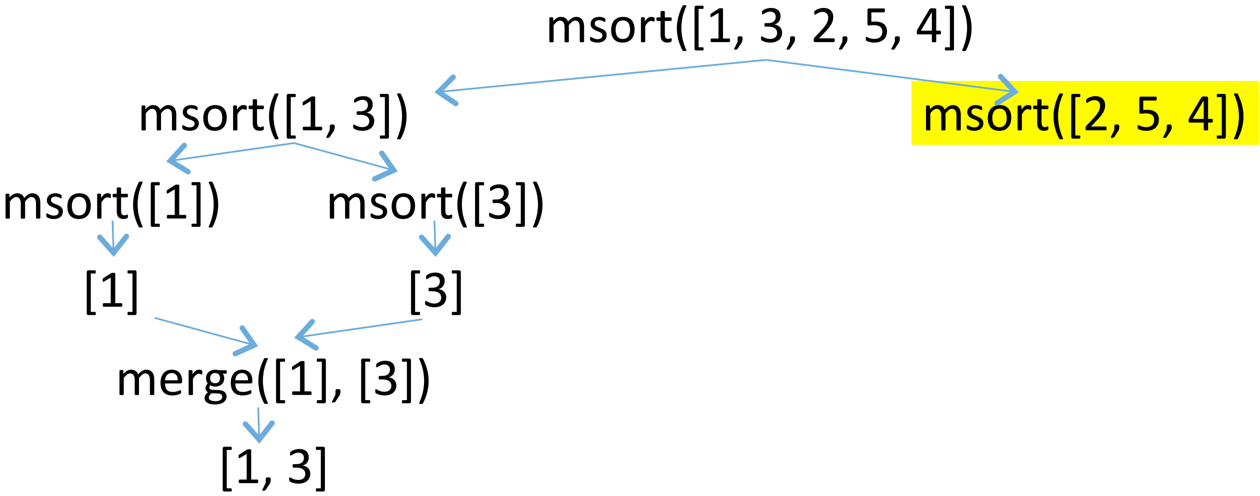
[1]

[3]

merge([1], [3])

[1, 3]

Mergesort: example



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

msort([1, 3])

msort([2, 5, 4])

msort([1])

msort([3])

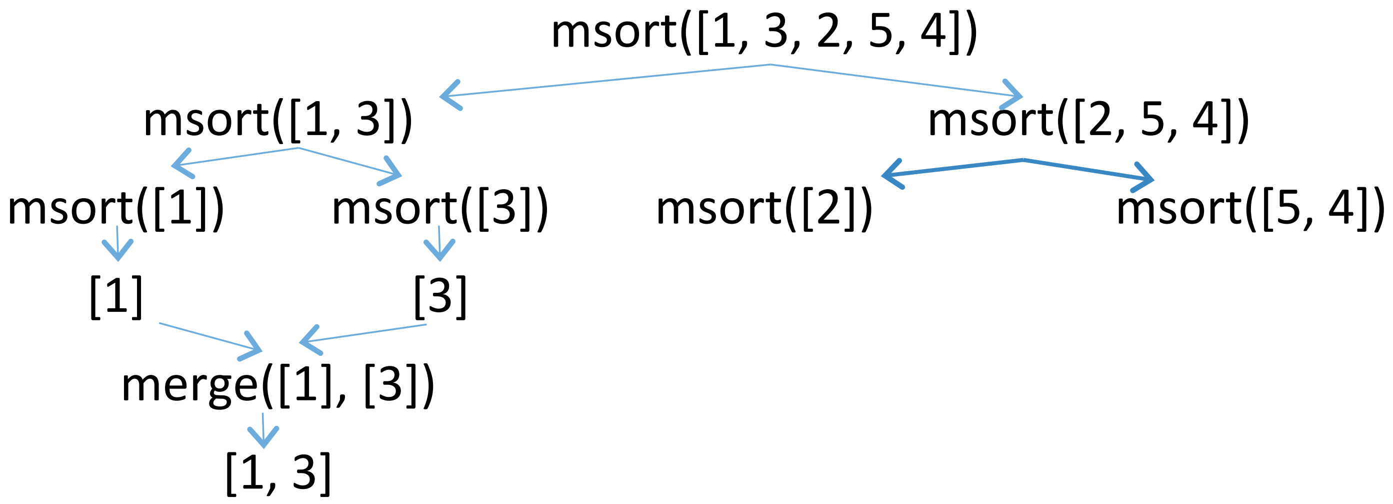
[1]

[3]

merge([1], [3])

[1, 3]

Mergesort: example



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

msort([1, 3])

msort([2, 5, 4])

msort([1])

msort([3])

msort([2])

msort([5, 4])

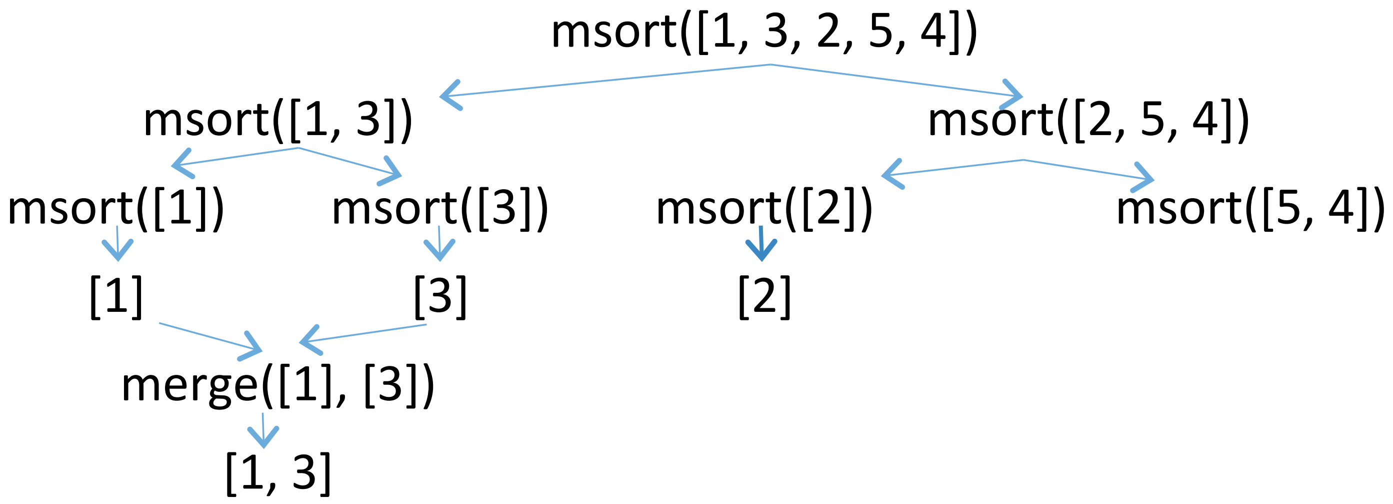
[1]

[3]

merge([1], [3])

[1, 3]

Mergesort: example



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

msort([1, 3])

msort([2, 5, 4])

msort([1])

msort([3])

msort([2])

msort([5, 4])

[1]

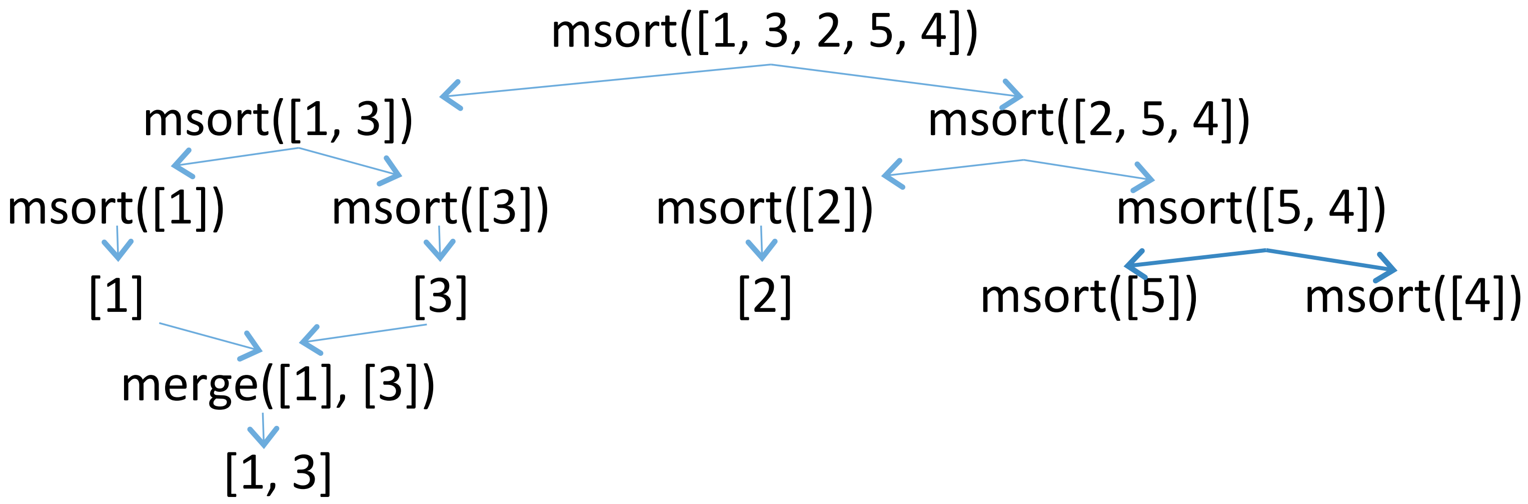
[3]

[2]

merge([1], [3])

[1, 3]

Mergesort: example



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

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

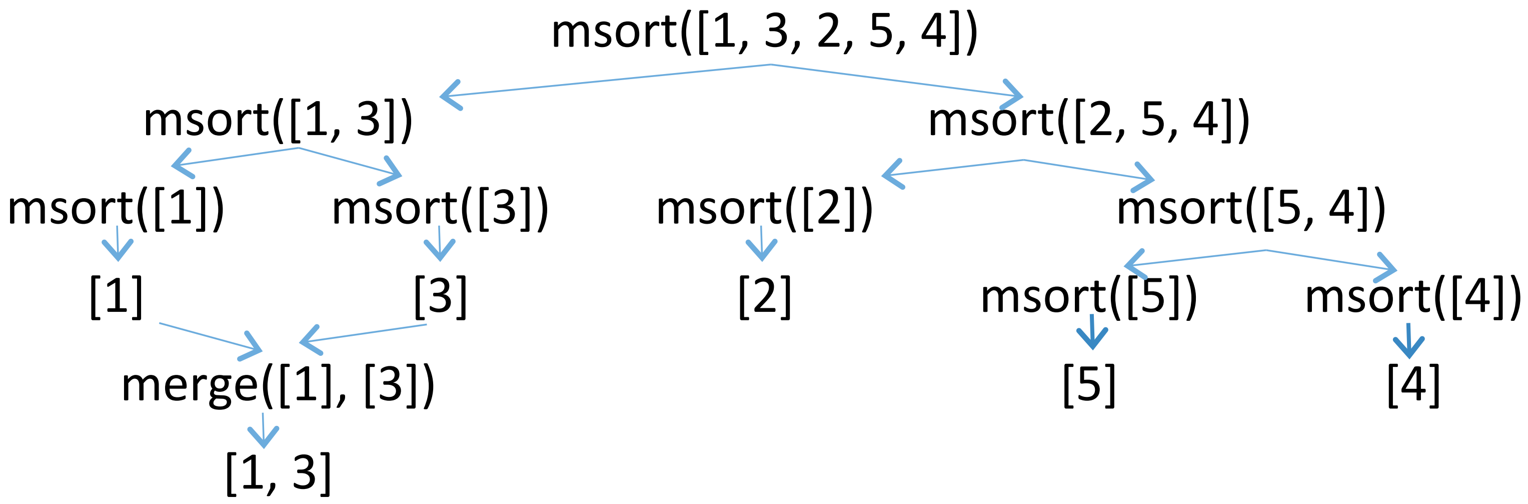
msort([1]) msort([3]) msort([2]) msort([5, 4])

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

merge([1], [3])

[1, 3]

Mergesort: example



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

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

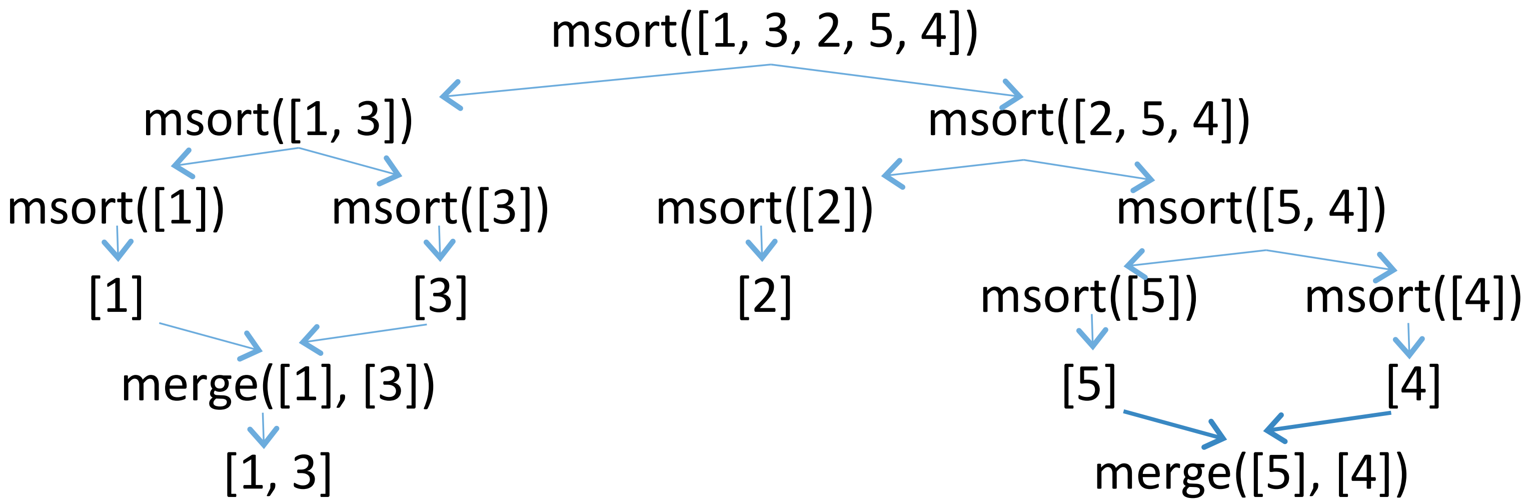
msort([1]) msort([3]) msort([2]) msort([5, 4])

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

merge([1], [3]) [5] [4]

[1, 3]

Mergesort: example



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

msort([1, 3])

msort([2, 5, 4])

msort([1])

msort([3])

msort([2])

msort([5, 4])

[1]

[3]

[2]

msort([5])

msort([4])

merge([1], [3])

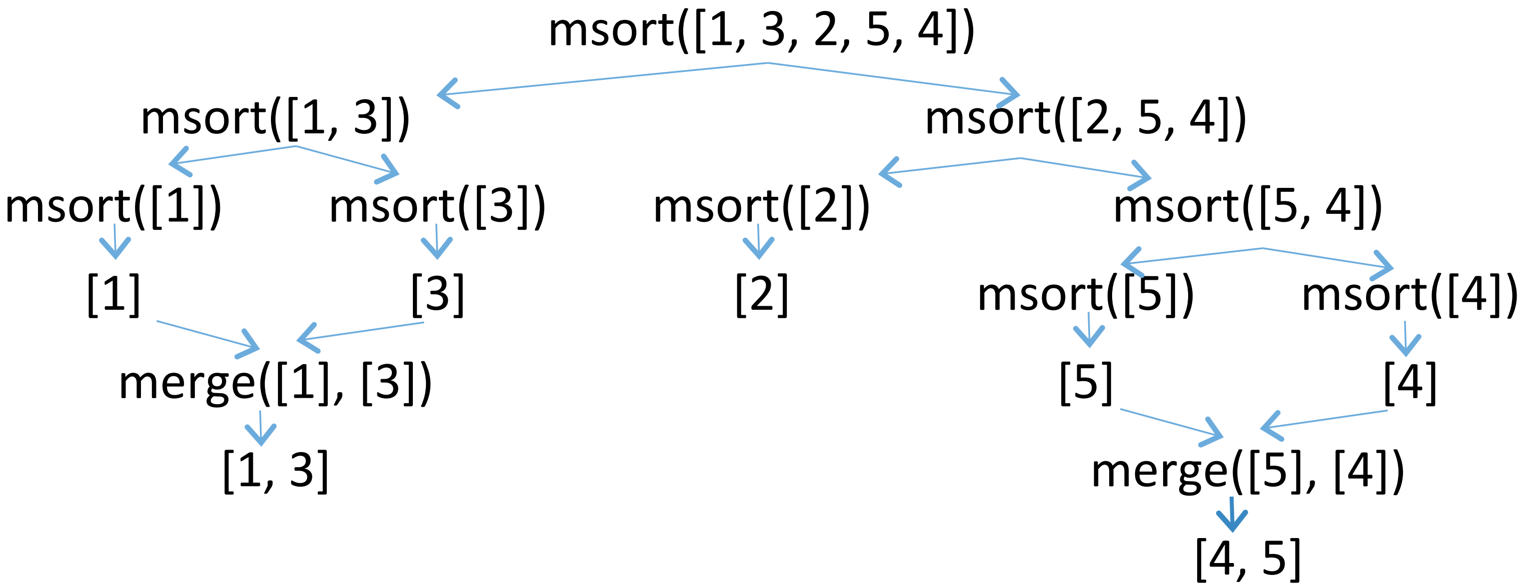
[5]

[4]

[1, 3]

merge([5], [4])

Mergesort: example



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

msort([1, 3])

msort([2, 5, 4])

msort([1])

msort([3])

msort([2])

msort([5, 4])

[1]

[3]

[2]

msort([5])

msort([4])

merge([1], [3])

[5]

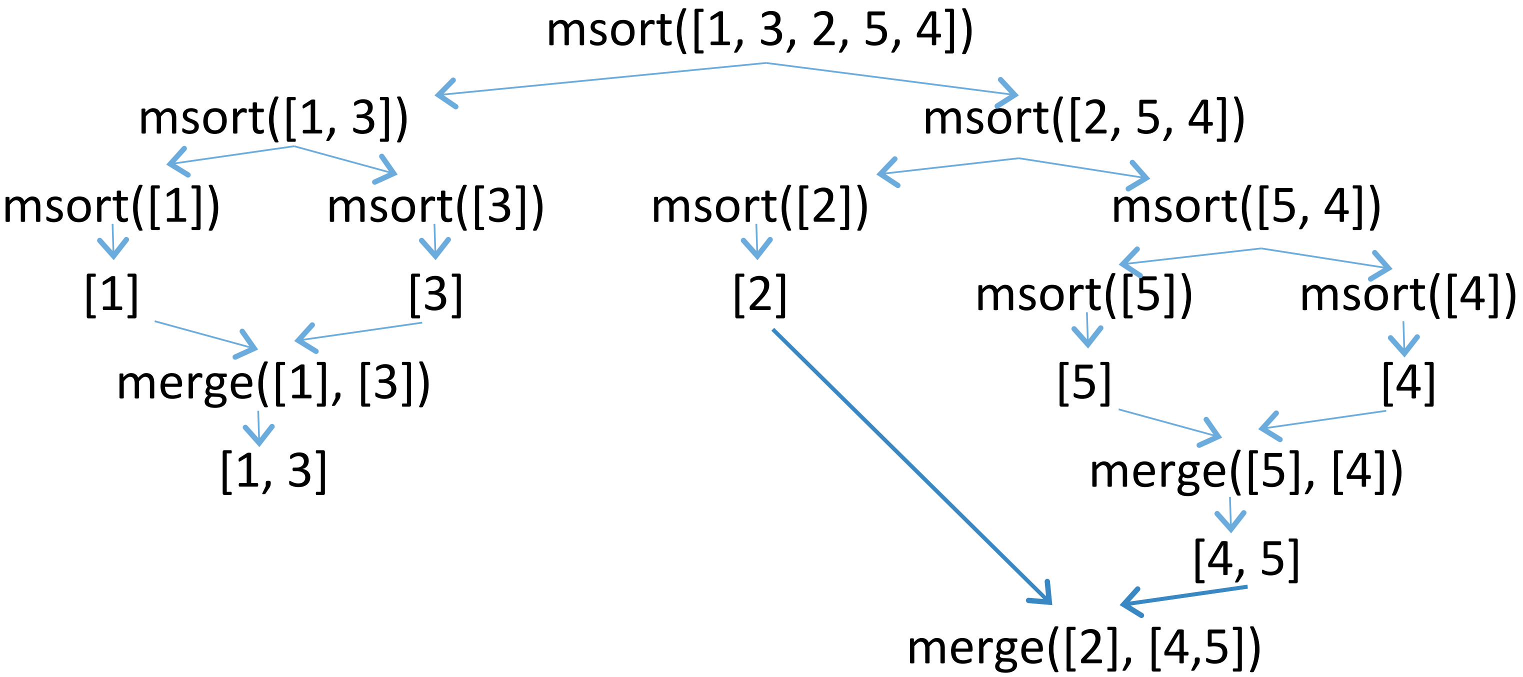
[4]

[1, 3]

merge([5], [4])

[4, 5]

Mergesort: example



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

msort([1, 3])

msort([2, 5, 4])

msort([1])

msort([3])

msort([2])

msort([5, 4])

[1]

[3]

[2]

msort([5])

msort([4])

merge([1], [3])

[5]

[4]

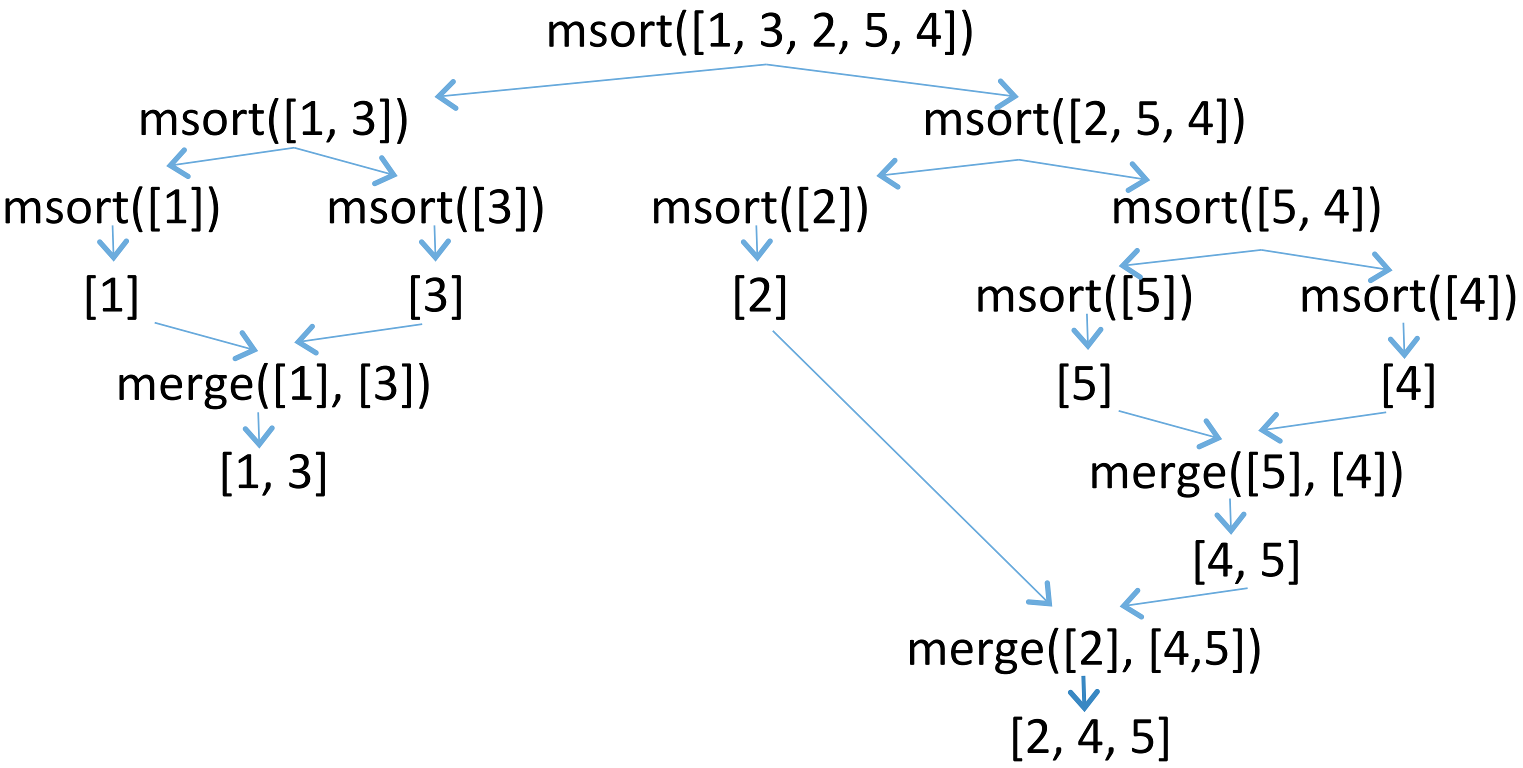
[1, 3]

merge([5], [4])

[4, 5]

merge([2], [4,5])

Mergesort: example



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

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

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

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

merge([1], [3]) [5] [4]

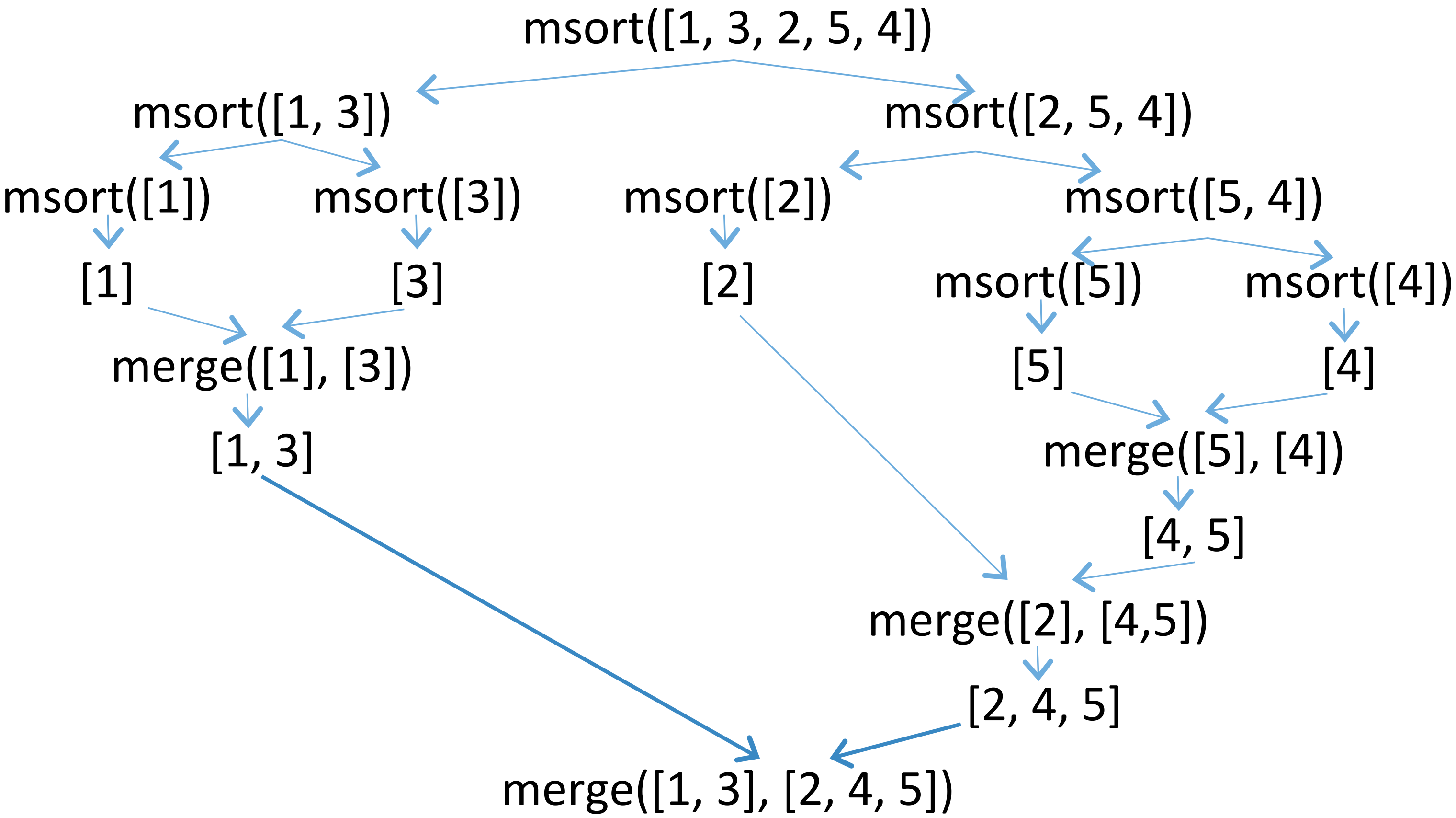
[1, 3] merge([5], [4])

[4, 5]

merge([2], [4,5])

[2, 4, 5]

Mergesort: example



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

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

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

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

merge([1], [3]) [5] [4]

[1, 3] merge([5], [4])

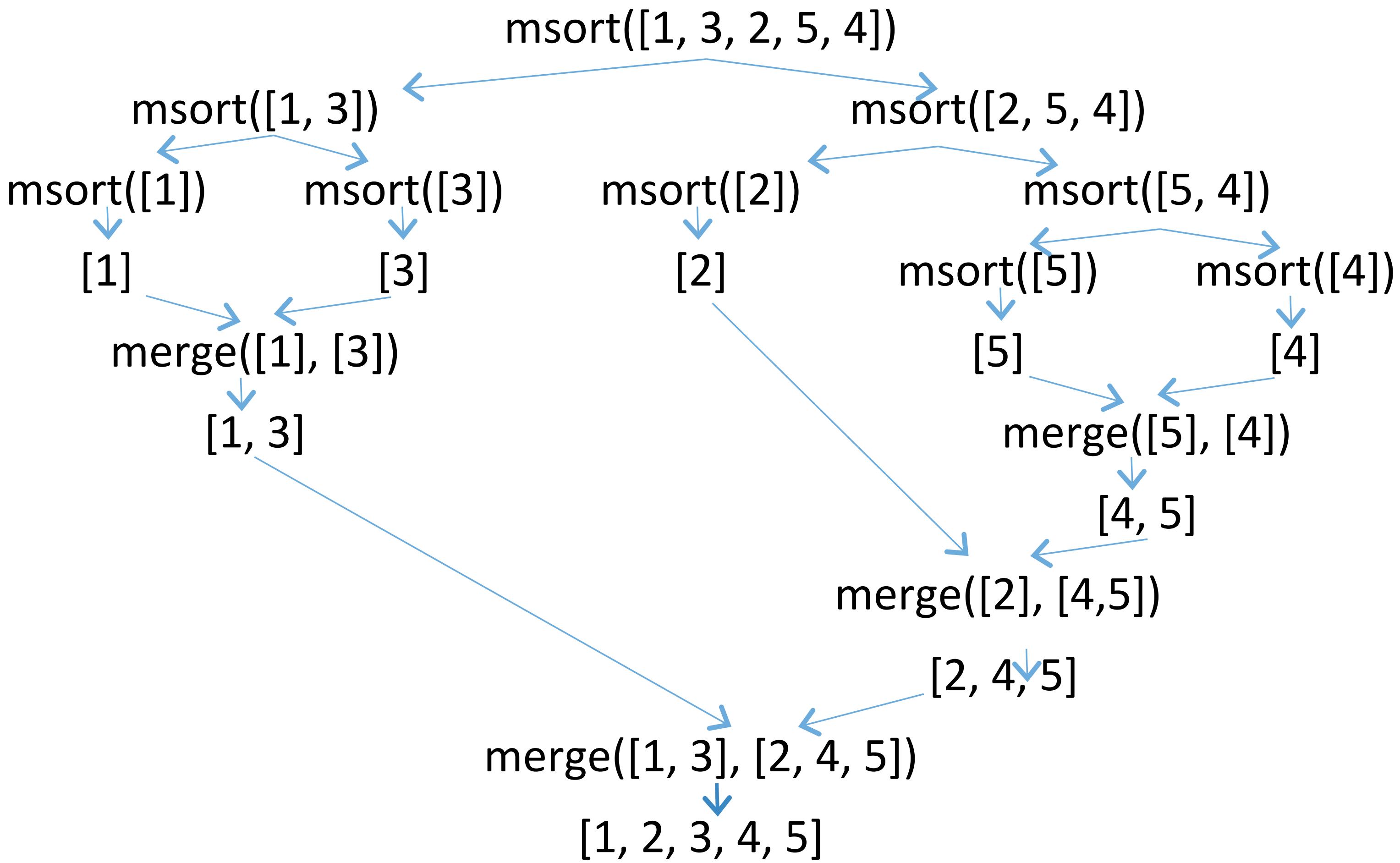
[4, 5]

merge([2], [4,5])

[2, 4, 5]

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

Mergesort: example



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

msort([1, 3])

msort([2, 5, 4])

msort([1])

msort([3])

msort([2])

msort([5, 4])

[1]

[3]

[2]

msort([5])

msort([4])

merge([1], [3])

[5]

[4]

[1, 3]

merge([5], [4])

[4, 5]

merge([2], [4,5])

[2, 4, 5]

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

[1, 2, 3, 4, 5]

Mergesort: complexity

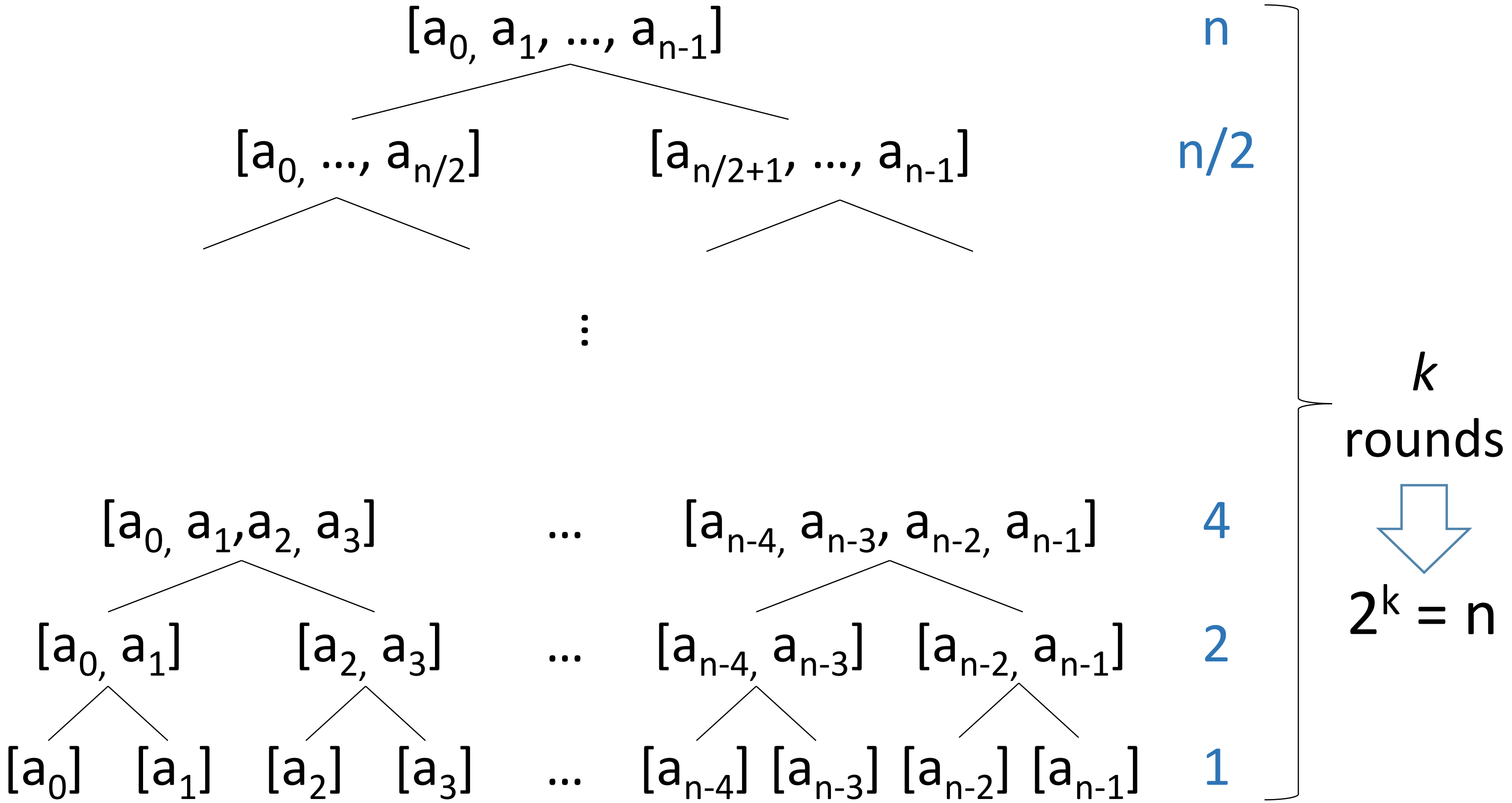
Cost = (cost per round of repetition) x (no. of rounds of repetition) worst case

merging the sorted lists is O(n)*

???

*if slicing is removed from merge

Mergesort: complexity



$[a_0, a_1, \ldots, a_{n-1}]$ n

$[a_0, \ldots, a_{n/2}]$ $[a_{n/2+1}, \ldots, a_{n-1}]$ $n/2$

⋮

$[a_0, a_1, a_2, a_3]$ … $[a_{n-4}, a_{n-3}, a_{n-2}, a_{n-1}]$ 4

$[a_0, a_1]$ $[a_2, a_3]$ … $[a_{n-4}, a_{n-3}]$ $[a_{n-2}, a_{n-1}]$ 2

$[a_0]$ $[a_1]$ $[a_2]$ $[a_3]$ … $[a_{n-4}]$ $[a_{n-3}]$ $[a_{n-2}]$ $[a_{n-1}]$ 1

k rounds

$2^k = n$

Mergesort: complexity

- No. of rounds of recursion:
  - if we start with a list of size n and have k rounds of recursion, then $2^k = n$

    $\therefore \log_2(2^k) = \log_2(n)$

    $\therefore k = \log_2(n)$

- Complexity of each round of recursion: O(n)

$\Rightarrow$ Worst-case complexity of mergesort: O(n log n)

recursion: summary

Recursion: summary

- Recursion offers a way to express repetitive computations cleanly and succinctly
- How to:
  - what are the values used in 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