(03) Iteration & Decomposition Part 2b

(Merge Sort)

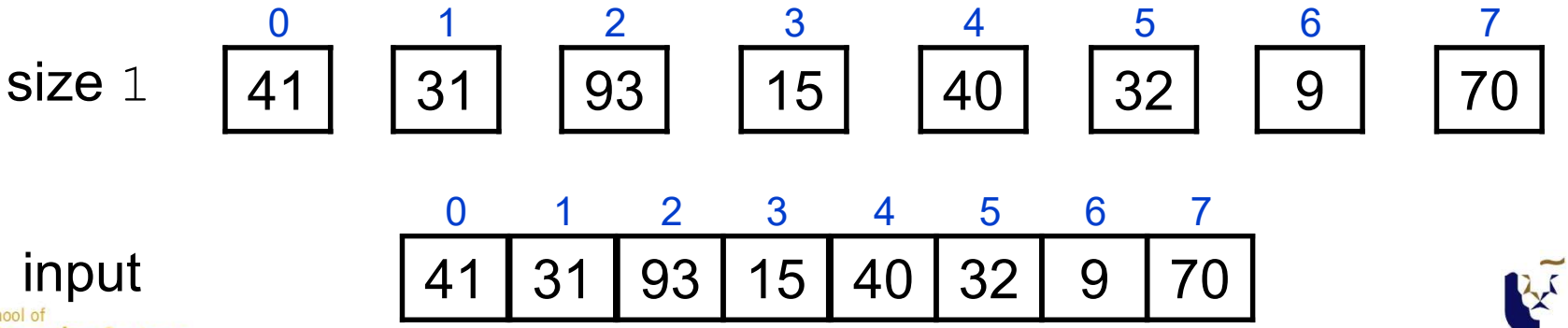
Video (11 mins):

https://www.youtube.com/watch?v=aSjCupRQvhY&list=PLi1cUmnkDnZvpLl1NPYxmq1Jnd7LAGCaa&index=33

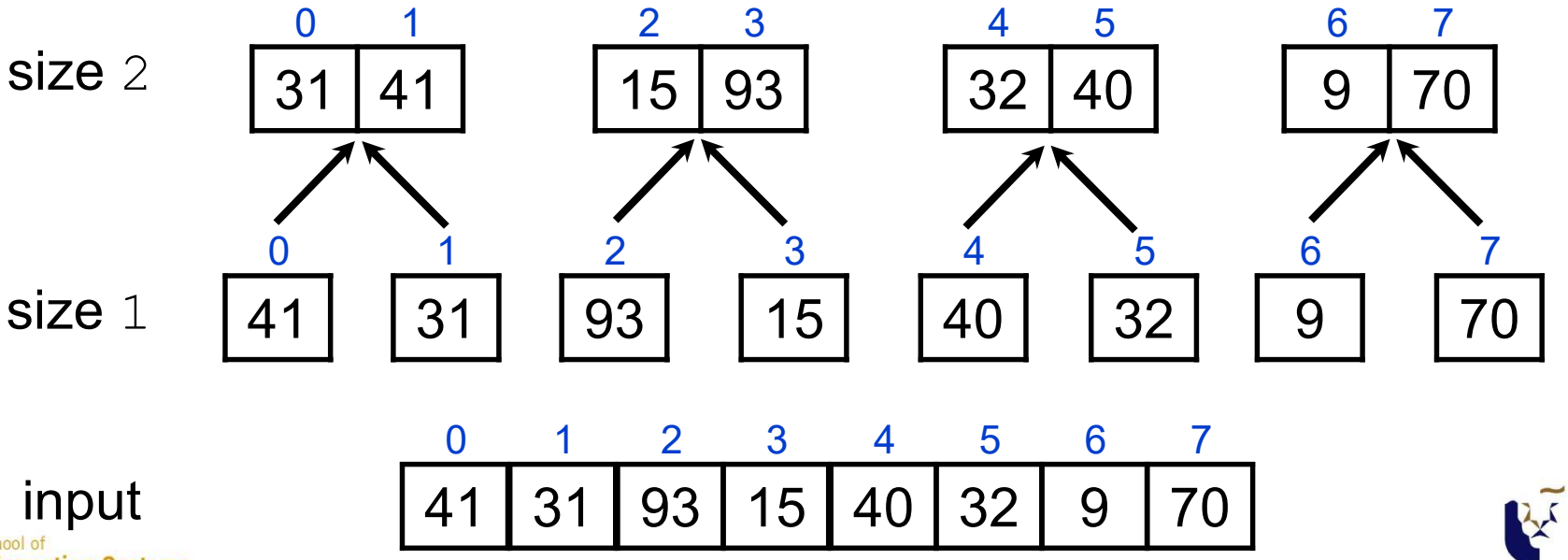
Merge Sort

- The merge sort algorithm works from “the bottom up”
  - start by solving the smallest pieces of the main problem
  - keep combining their results into larger solutions
  - eventually the original problem will be solved

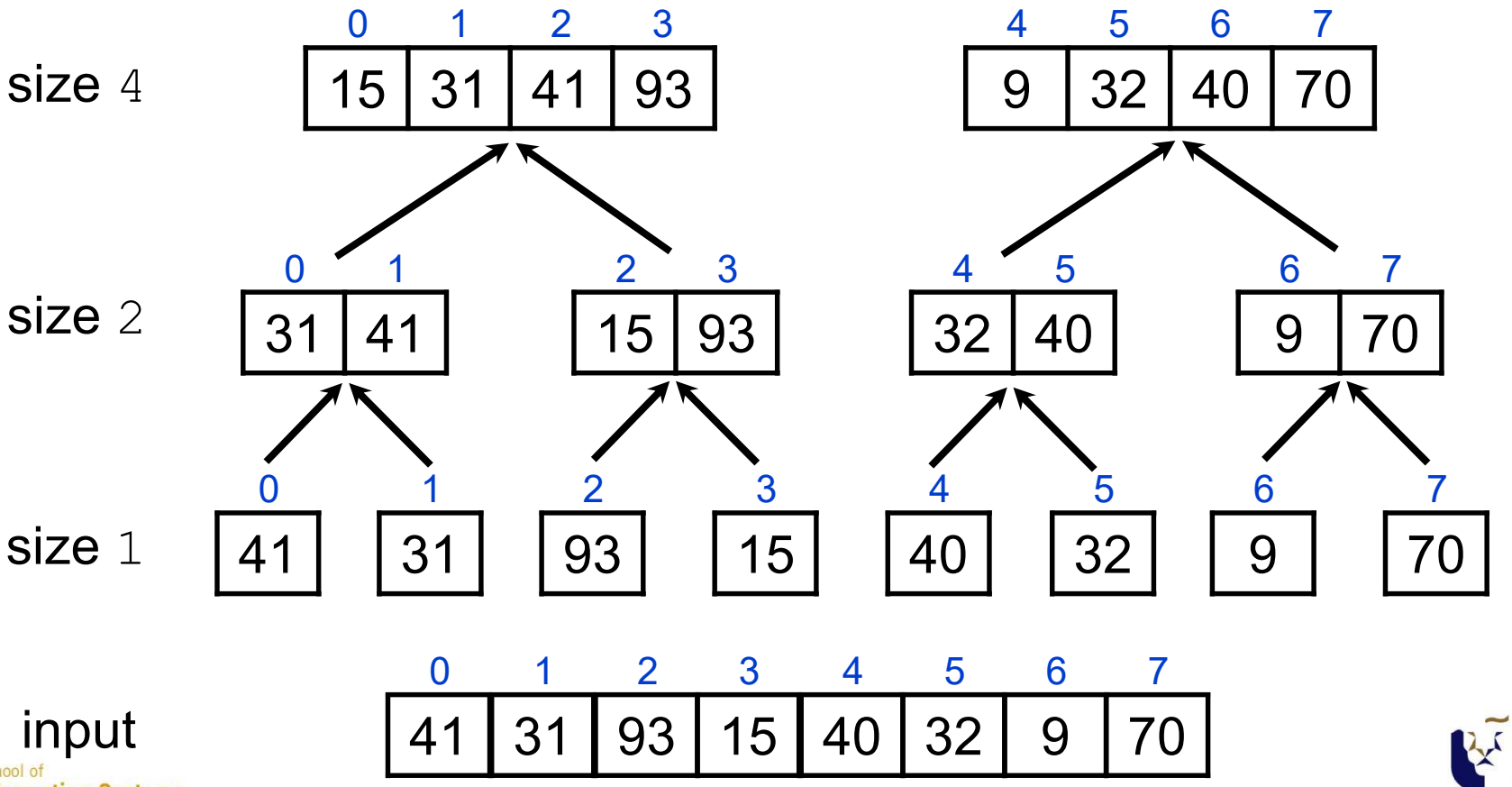
Merge sort: Bottom-up Approach



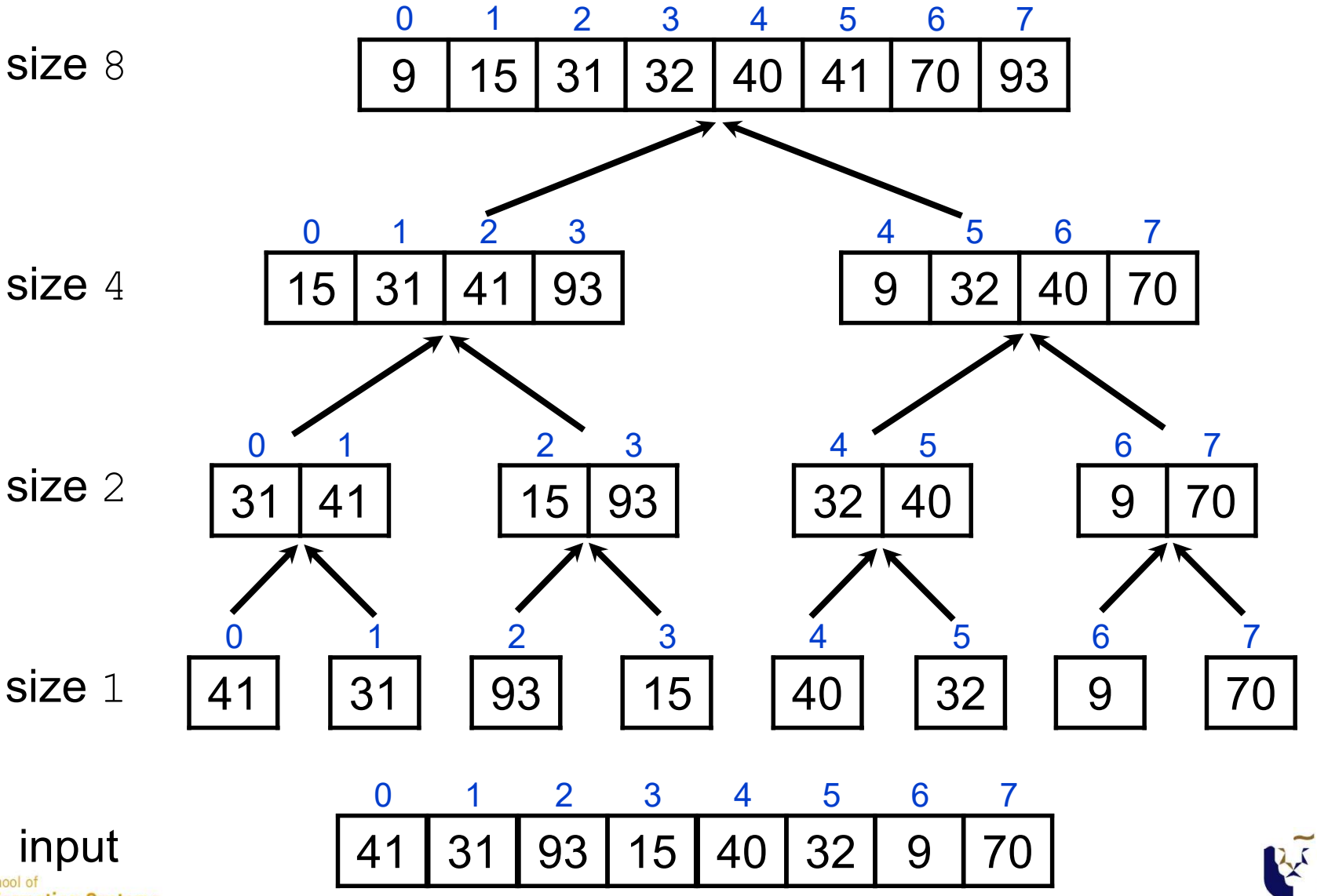
Merge sort: Bottom-up Approach



Merge sort: Bottom-up Approach



Merge sort: Bottom-up Approach



mSort

- The merge sort algorithm has been implemented in **SearchSortAnimation** as a function named `mSort`
- What you should know:
  - the group size (**gs**) is initialized to 1 and doubles at each successive level
  - within a level, pairs of groups are identified
  - a helper function named `merge` does the hard work
- The first statement in the main loop is on line 4
  - we'll attach a probe here to look at the array at the start of each iteration
  - a special version of `brackets` will draw pairs of brackets around each group

1st Loop: Iterate Over Increasing Group Sizes

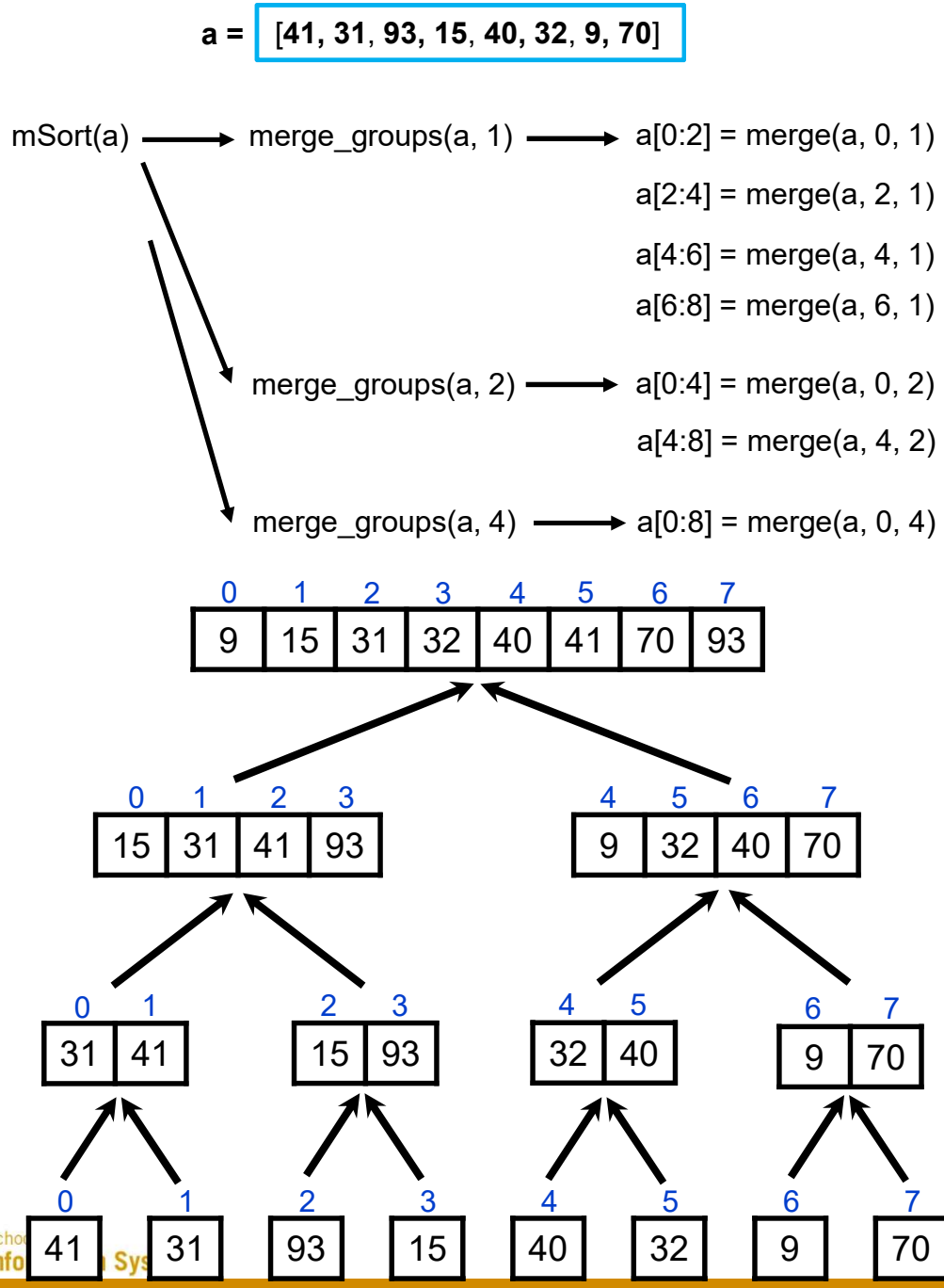
```
1:  def mSort(array):
2:    groupsize = 1                  #start from groups of size 1
3:    while groupsize < len(array):
4:      merge_groups(array, groupsize) #merge pairs of groups
5:      groupsize *= 2                 #double the size of the group
6:    return array
```

- Every iteration doubles the group size
  - size = 1, size = 2, size = 4, ..., size = `n`
- After `k` steps, we will cover an array of size $2^k = n$
  - The number of steps $k = \log_2 n$
- This is just the 1st loop. There is another loop inside `merge_groups`.

2nd Loop: Iterate Over Pairs of Groups of a Given Size

```
1:  def merge_groups(a, gs)
2:    i = 0                          # 1st group starts here
3:    while i < len(a):
4:      j = i + 2*gs                 # end of 2nd group
5:      a[i:j] = merge(a, i, gs)     # merge groups at a[i] and
                                       a[i+gs]
6:      i += 2*gs                    # next groups starts 2*gs
                                       places to the right
```

- In each iteration, we merge a pair of groups
- In an array of size `n`, how many pairs of groups are there of size `gs`?
  - if n = 8 and gs = 2, there are 4 groups or 2 pairs of groups
  - The number of steps is the number of pairs is `n` / (2 x `gs`)
- There is a third loop inside of `merge`.

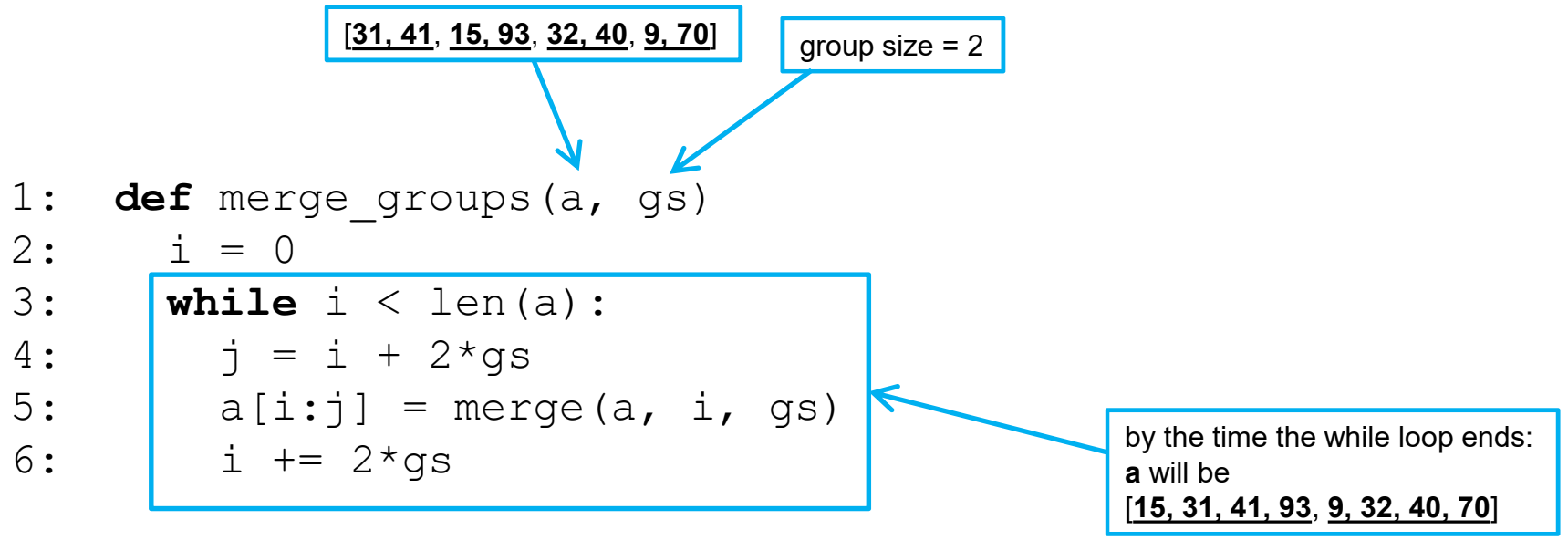



```
1:  def mSort(array):
2:    groupsize = 1
3:    while groupsize < len(array):
4:      merge_groups(array, groupsize)
5:      groupsize *= 2
6:    return array
```

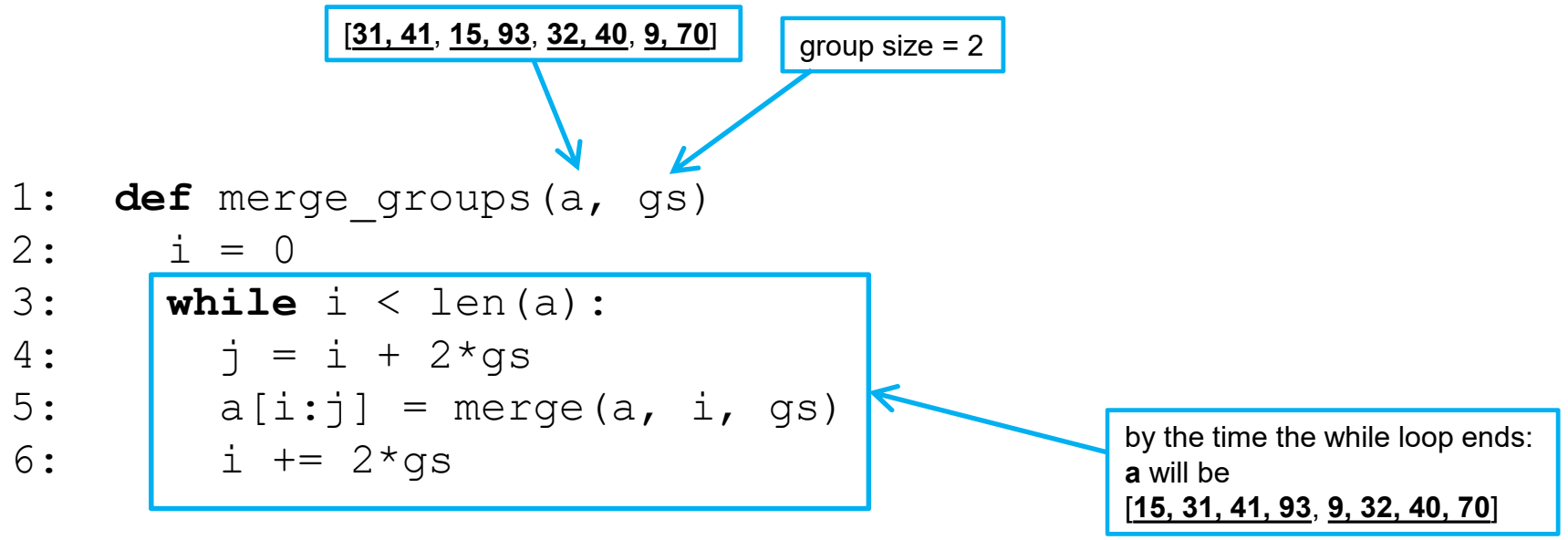
```
1:  def merge_groups(a, gs)
2:    i = 0
3:    while i < len(a):
4:      j = i + 2*gs
5:      a[i:j] = merge(a, i, gs)
6:      i += 2*gs
```

```
def merge(array, i, groupsize):
    r = []
    firstGroup = array[i:i+groupsize]
    secondGroup = array[i+groupsize:i+groupsize*2]

    while (len(firstGroup) != 0 or len(secondGroup) != 0):
        if(len(firstGroup) == 0):
            while (len(secondGroup) != 0):
                r.append(secondGroup.pop(0))
        elif (len(secondGroup) == 0):
            while (len(firstGroup) != 0):
                r.append(firstGroup.pop(0))
        else:
            if(firstGroup[0] > secondGroup[0]):
                r.append(secondGroup.pop(0))
            else:
                r.append(firstGroup.pop(0))
    return r
```

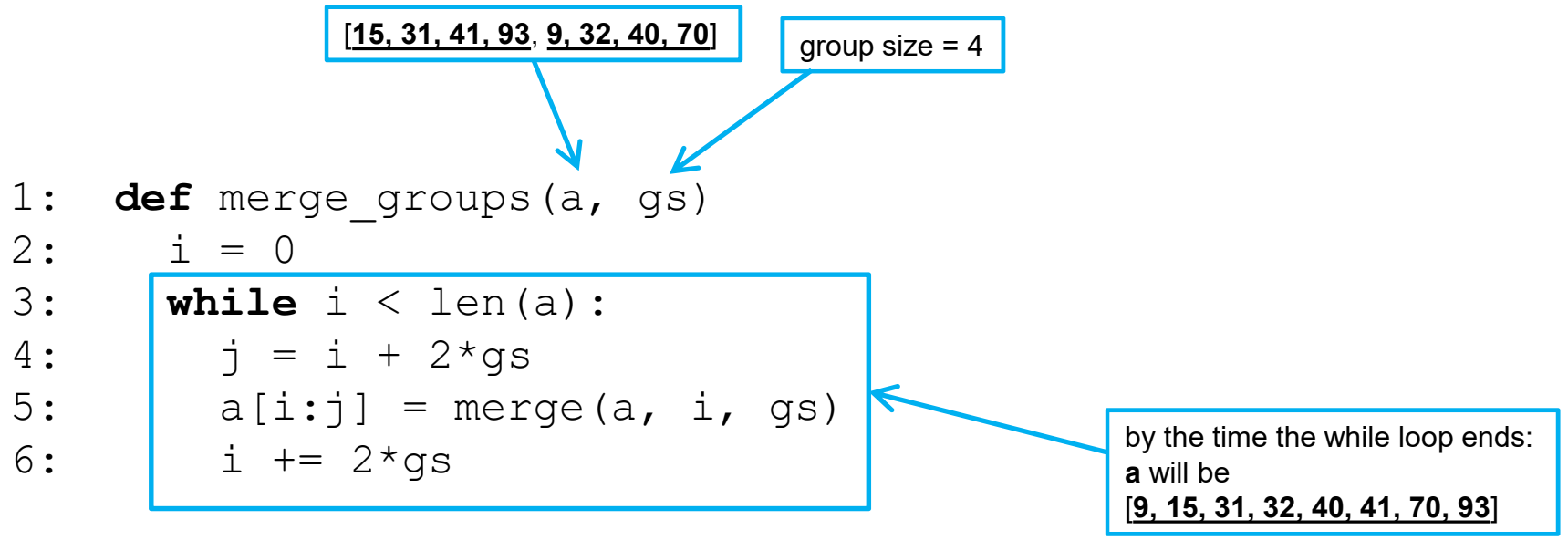
Example: merge_groups



Example: merge_groups



Example: merge_groups



3rd Loop: Iterate Over Items in Each Pair of Groups

```
merge function:
1  create a new result array r to store the sorted result
2  while there are still items in the 1st or 2nd group
       compare the top items in the 2 groups
       if the top item in the 1st group is smaller
           move top item from 1st group into r
       else
           move top item from 2nd group into r
3  return r
```

- In each iteration, we compare 2 items and move one item to result
- In the worst case, we have to do a comparison for every move
  - We can have fewer comparisons than moves if one group runs out of items first, then we simply move items from the other group without comparison
- For group size `gs`, the maximum number of moves is the maximum number of items in the pair of groups
  - The max number of steps is 2 x `gs`.

3rd Loop: Iterate Over Items in Each Pair of Groups

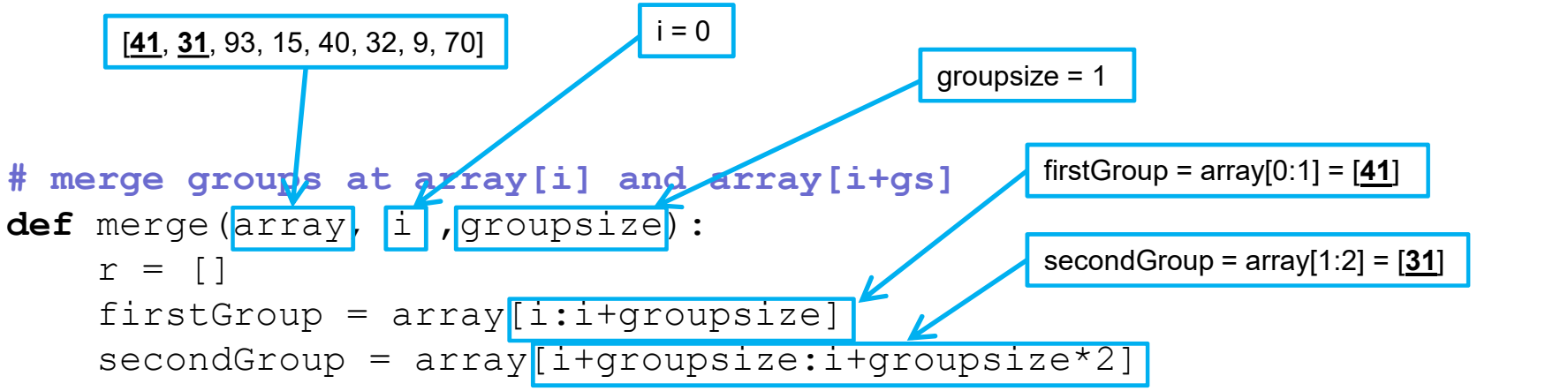
```
# merge groups at array[i] and array[i+gs]
def merge(array, i, groupsize):
1   r = []                              # new array to store merged result
    firstGroup = array[i:i+groupsize]
    secondGroup = array[i+groupsize:i+groupsize*2]

2   while (len(firstGroup) != 0 or len(secondGroup) != 0):
        if(len(firstGroup) == 0):
            while (len(secondGroup) != 0):
                r.append(secondGroup.pop(0))
        elif (len(secondGroup) == 0):
            while (len(firstGroup) != 0):
                r.append(firstGroup.pop(0))
        else:
            if(firstGroup[0] > secondGroup[0]):
                r.append(secondGroup.pop(0))
            else:
                r.append(firstGroup.pop(0))

3   return r
```

if 1st group is empty, just append all the remaining elements in 2nd group to r

if 2nd group is empty, just append all the remaining elements in 1st group to r

if there are elements in both groups, remove the smaller one from either group & append that to r

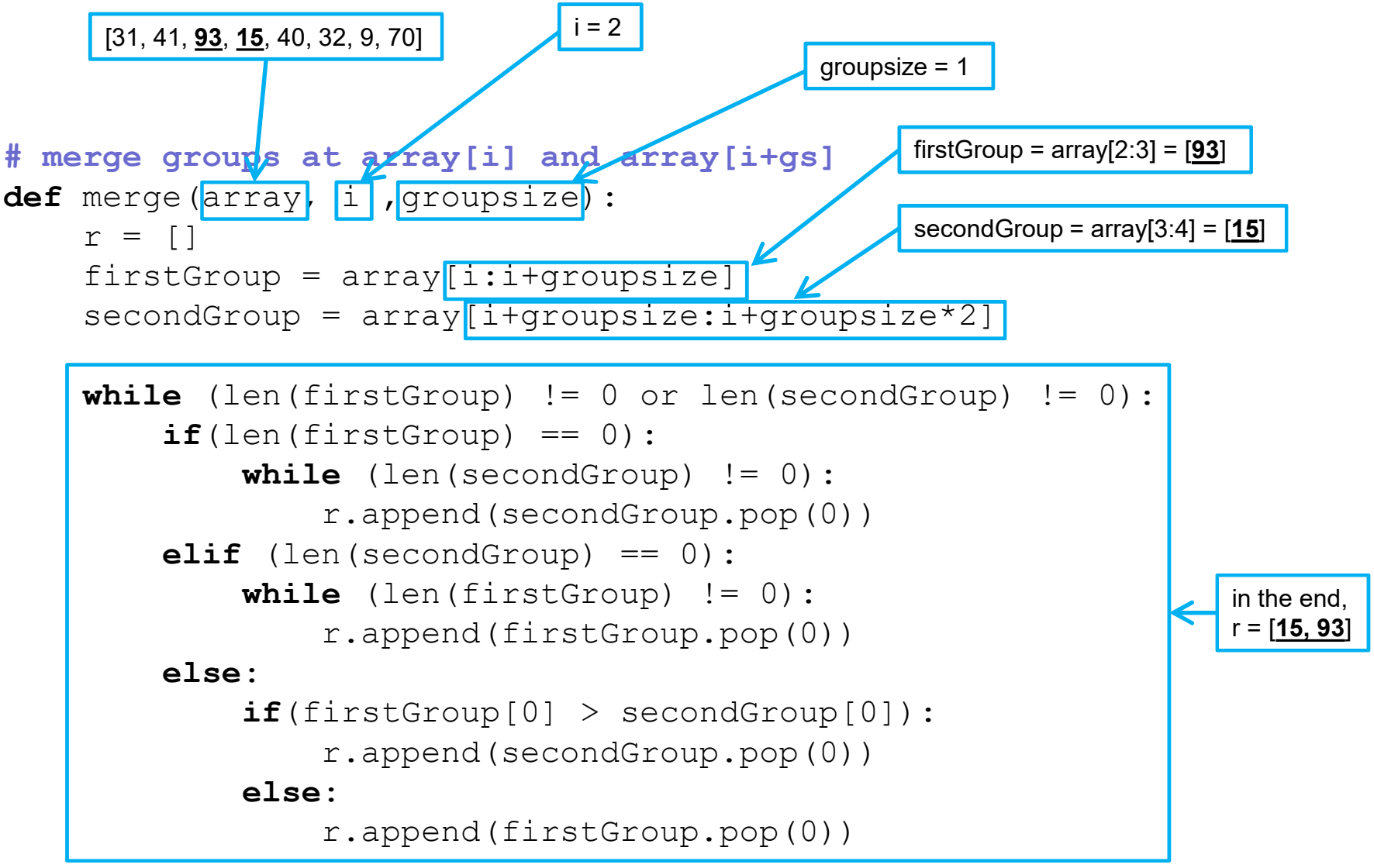
Example: merge




```
# merge groups at array[i] and array[i+gs]
def merge(array, i ,groupsize):
    r = []
    firstGroup = array[i:i+groupsize]
    secondGroup = array[i+groupsize:i+groupsize*2]

    while (len(firstGroup) != 0 or len(secondGroup) != 0):
        if(len(firstGroup) == 0):
            while (len(secondGroup) != 0):
                r.append(secondGroup.pop(0))
        elif (len(secondGroup) == 0):
            while (len(firstGroup) != 0):
                r.append(firstGroup.pop(0))
        else:
            if(firstGroup[0] > secondGroup[0]):
                r.append(secondGroup.pop(0))
            else:
                r.append(firstGroup.pop(0))

    return r
```

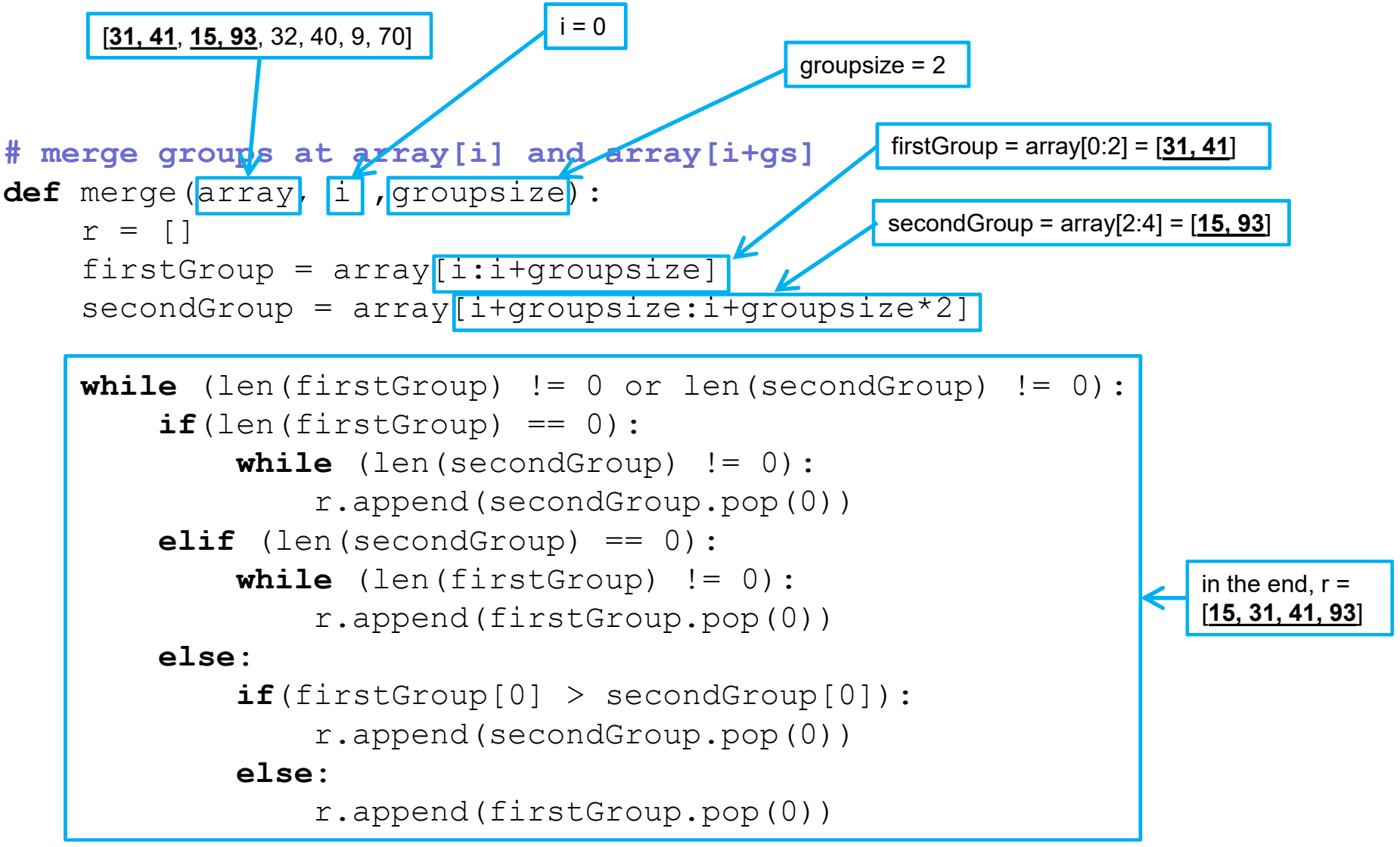
in the end, r = [31, 41]

Example: merge




```
# merge groups at array[i] and array[i+gs]
def merge(array, i ,groupsize):
    r = []
    firstGroup = array[i:i+groupsize]
    secondGroup = array[i+groupsize:i+groupsize*2]

    while (len(firstGroup) != 0 or len(secondGroup) != 0):
        if(len(firstGroup) == 0):
            while (len(secondGroup) != 0):
                r.append(secondGroup.pop(0))
        elif (len(secondGroup) == 0):
            while (len(firstGroup) != 0):
                r.append(firstGroup.pop(0))
        else:
            if(firstGroup[0] > secondGroup[0]):
                r.append(secondGroup.pop(0))
            else:
                r.append(firstGroup.pop(0))

    return r
```

Example: merge

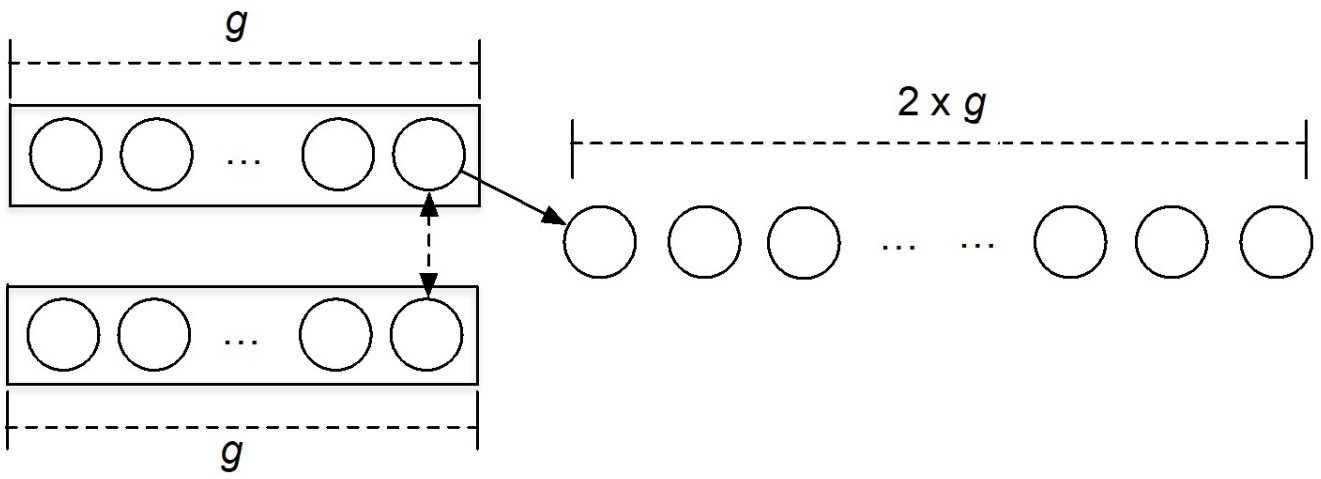


Animated Demo: `mSort`

- An example of how to call `mSort` animation

```
>> from SearchSortAnimation import *

>> arr = [5, 9, 8, 2, 11, 3, 6, 1]

>> mSort(arr, True)
```

“Merge” Operation at Level g



(2 * *g*) -1 comparisons are needed to merge 2 sorted lists with *g* elements each (in the worst case).
(2 * g) moves (appends) are needed to merge these 2 lists.

e.g. Try to merge these pairs of lists:

(a) [1, 2, 3, 4], [5, 6, 7, 8]

(b) [1, 3, 5, 7], [2, 4, 6, 8]

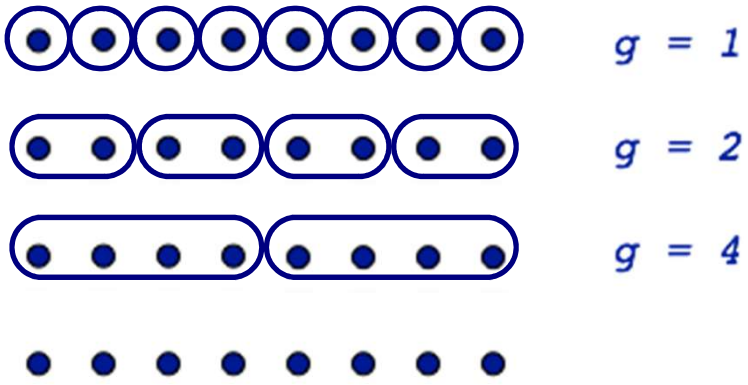
How many comparisons are required in each case?

Complexity of Merge Sort

- No. of moves required:
  - no. of pairs to be merged at level g = $\frac{n}{2 * g}$
  - no. of moves needed to merge a pair at level g = **(2 * g)**

→ no. of moves at level g =

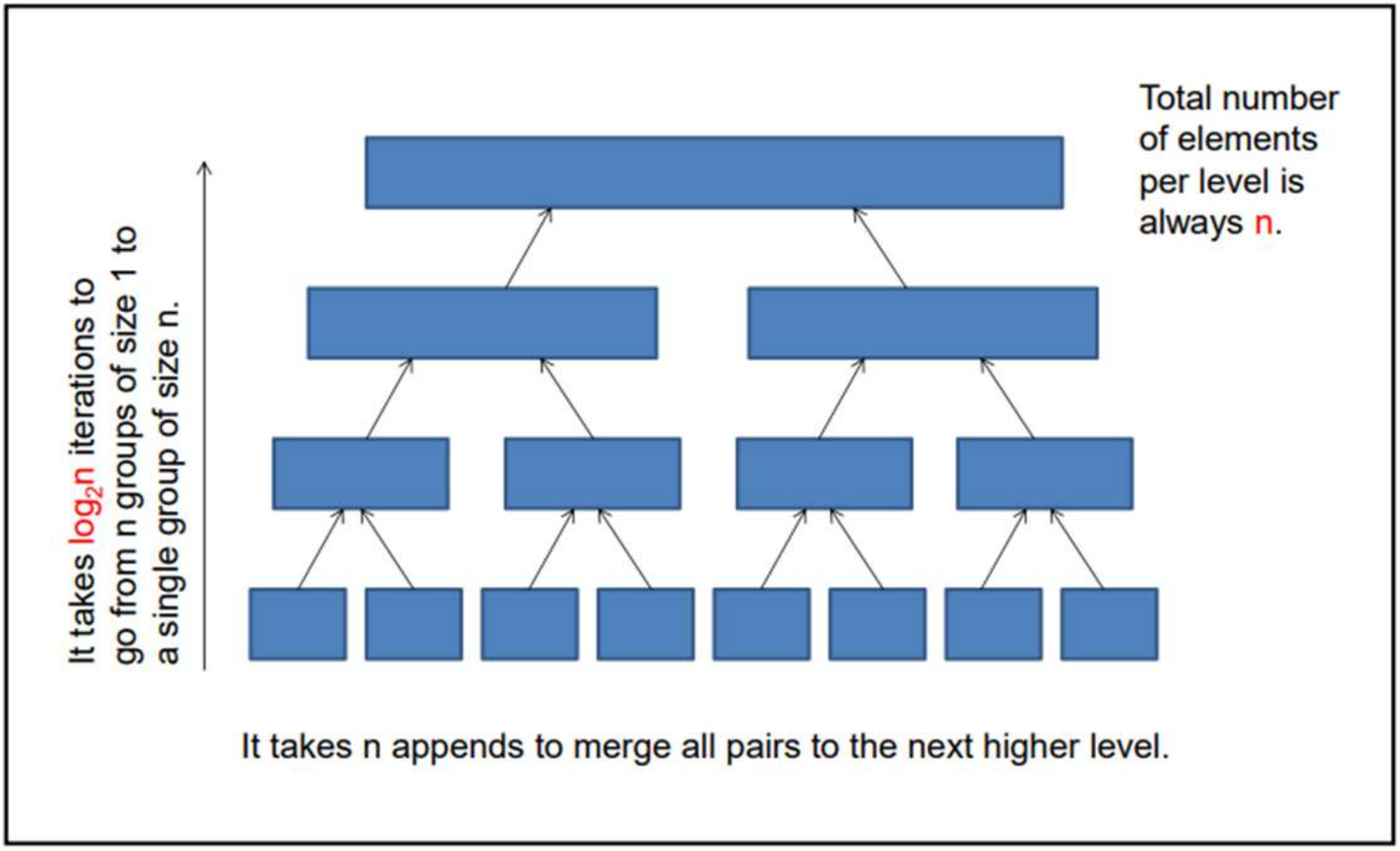
$\frac{n}{2 * g} * (2 * g)$ = n



- Like binary search, it takes **($\log_2 n$)** levels to go from 1 to n

  → total no. of steps = n * $\log_2$n

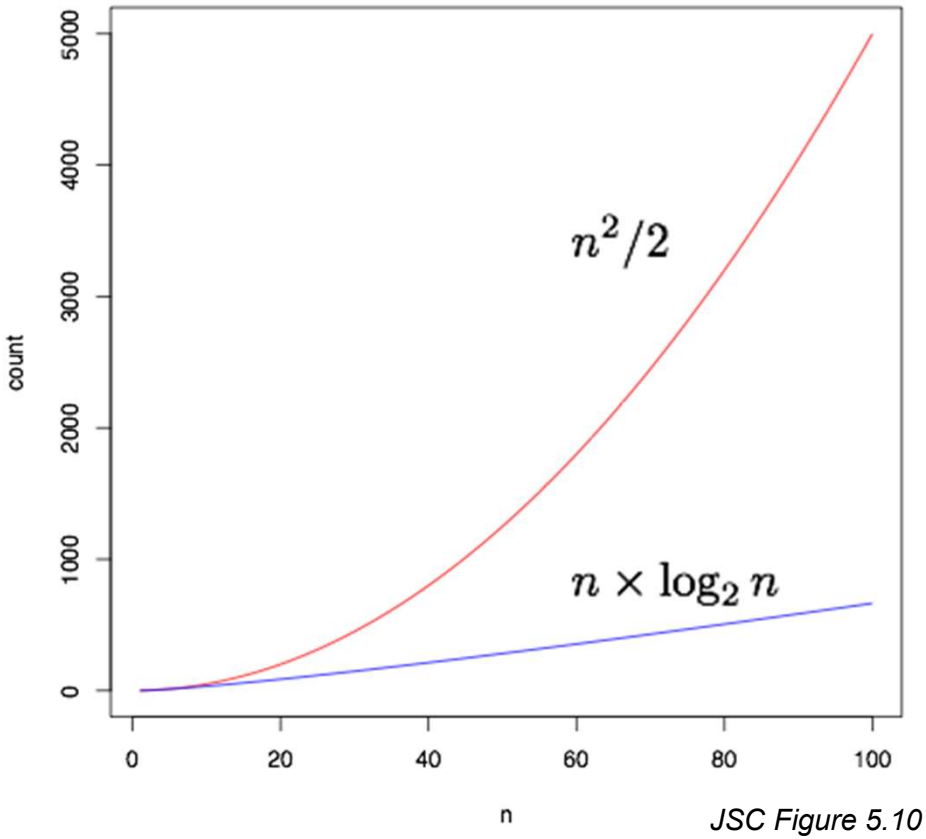
- Complexity is $O(n \log n)$



Taken from: https://www.cs.cmu.edu/~15110-f12/Unit05PtC-handout.pdf

Scalability of Merge Sort

- Is this new formula that much better than the $n^2 / 2$ comparisons made by `iSort`?
  - not that big of a difference for small arrays
  - but for larger arrays the difference is clear



JSC Figure 5.10

In-Class Exercises

For each of the array below, which sorting algorithm: insertion sort or merge sort, will make **more** comparisons?

(a) When `x = [93, 85, 22, 69, 73, 59]`

(b) When `x = [24, 56, 30, 28, 47, 91, 60, 36]`

(04) Iteration & Decomposition

Solution to In-class Exercise

Video (15 mins):

https://www.youtube.com/watch?v=BzZip5pUqdw&list=PLi1cUmnkDnZvpLI1NPYxmq1Jnd7LAGCaa&index=34

Recap: Sort

- The insertion sort algorithm relies on iteration
  - uses nested loops
  - each time moving an element leftward to its rightful position
  - Complexity: $O(n^2)$

- The merge sort algorithm uses divide and conquer
  - systematically divide the problem into smaller sub-problems
  - Complexity: $O(n \log n)$

Summary

- The concepts of iteration and decomposition
- Illustrated through two problems: searching and sorting
- Searching: linear search vs. binary search
- Sorting: insertion sort vs. merge sort

Road Map

Algorithm Design and Analysis

- Week 1: Introduction, Counting, Programming
- Week 2: Programming
- Week 3: Complexity
- Week 4: Iteration & Decomposition

This week: We explained merge sort using an iterative algorithm.

Next week → Week 5: Recursion

Next week: We will revisit merge sort using a recursive algorithm.

Fundamental Data Structures

(Weeks 6 - 10)

Computational Intractability and Heuristic Reasoning

(Weeks 11 - 13)