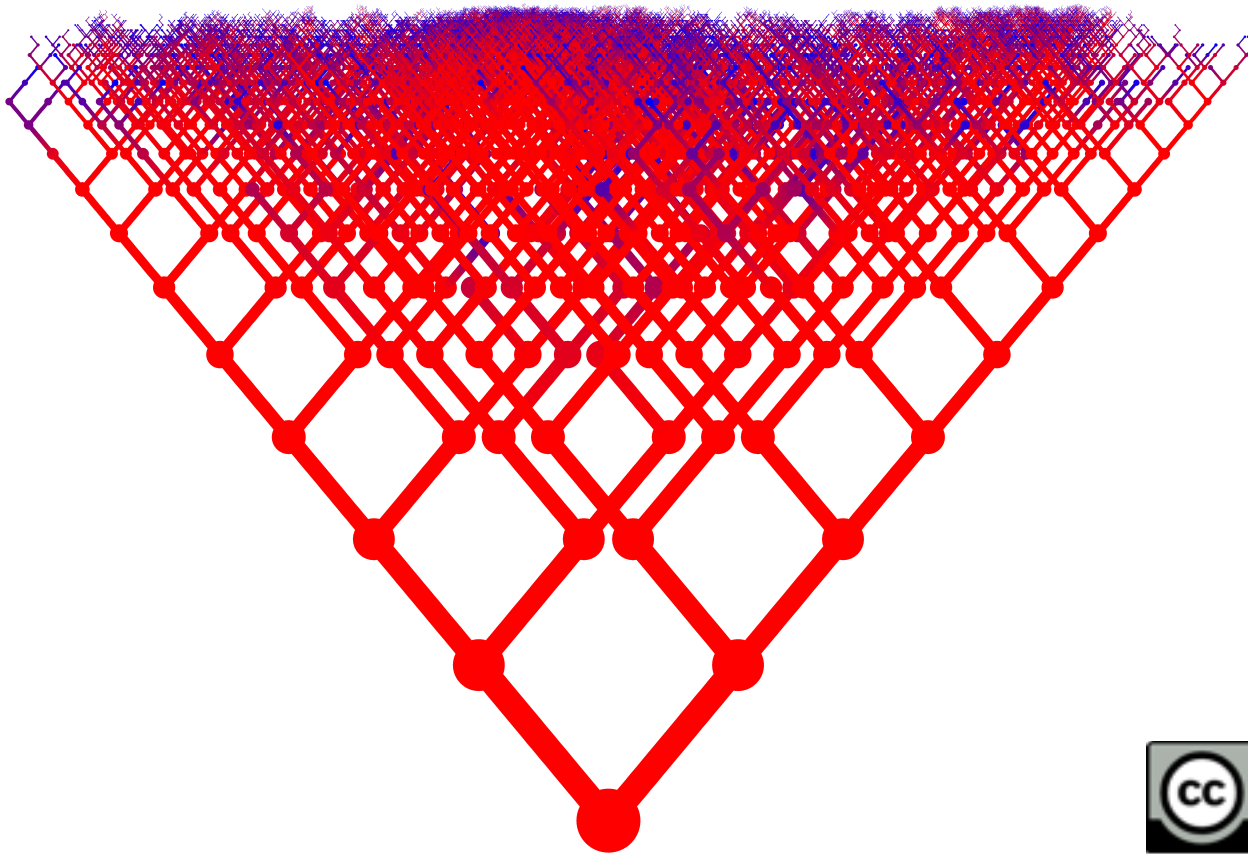


# Binary Search and Merge Sort



# Announcements

- Today:
  - Lab 6
  - Programming Assignment 5 (
- Tomorrow: Problem Set 5
- Python Tutors
- Exam on Thursday: Units 1 – 5 (inclusive)

# Today

- Recursion for search:
  - Binary Search
  - Merge Sort
- Logarithmic worst-case complexity

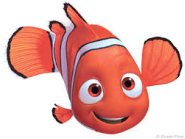


# Binary Search

# Thinking linearly...



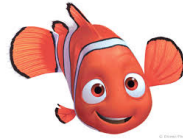
?



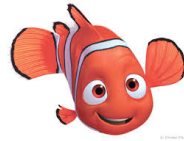
# Thinking linearly...



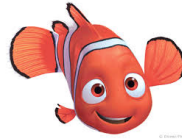
?



# Thinking linearly...



!



# Linear Search

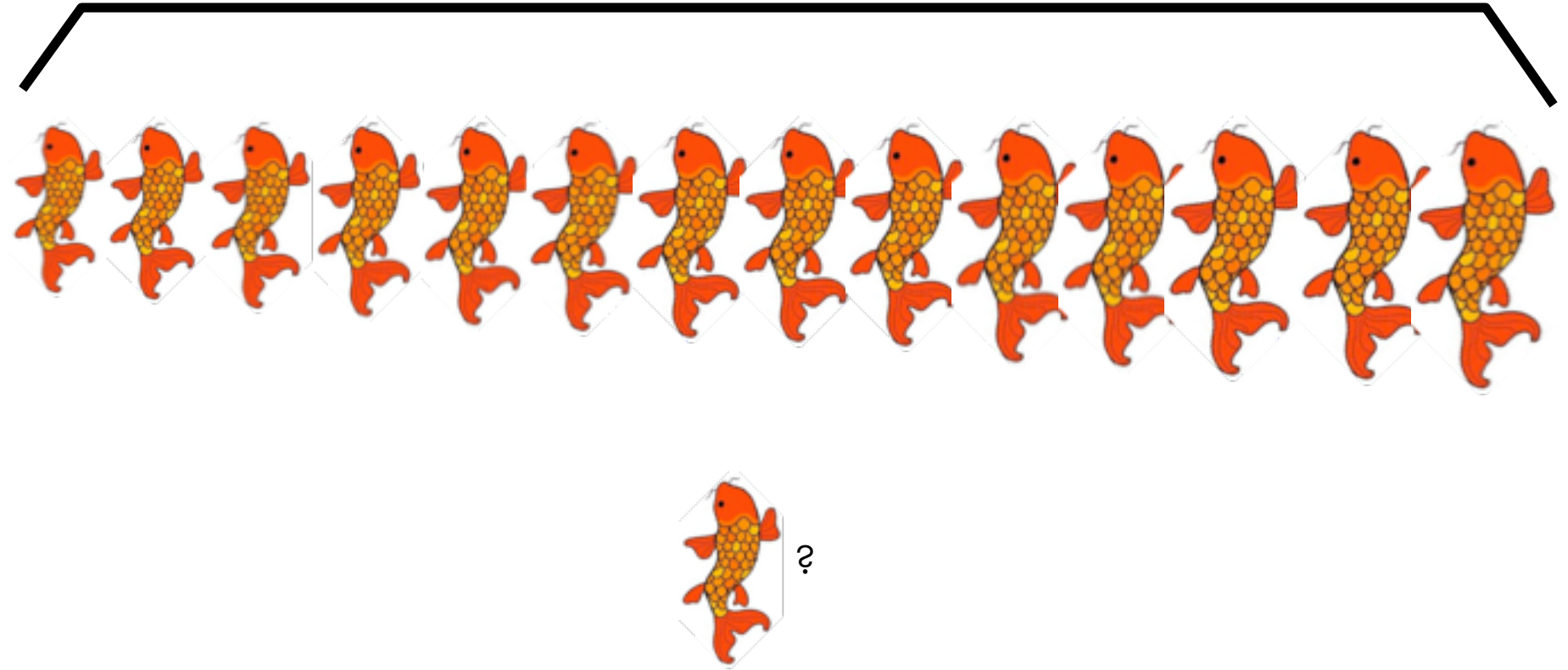
```
def contains(items, key):  
    for index in range(len(items)):  
        if items[index] == key:  
            return True  
    return False
```



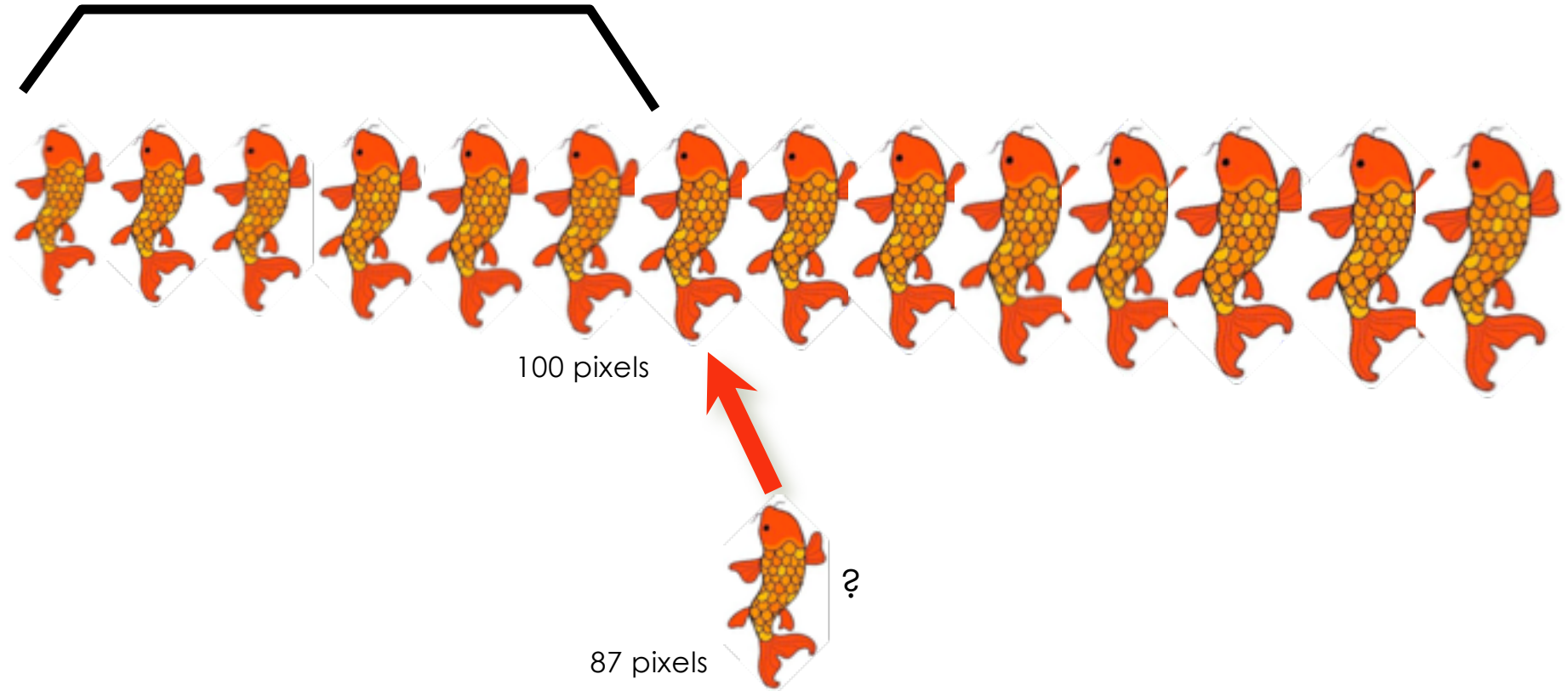
# Number guessing

- ▣ I'm thinking of a number between 1 and 16
- ▣ You get to ask me yes/no & is it greater than some number you choose
- ▣ How many questions do you need to ask?
- ▣ Which questions will you ask to get the answer quickest?

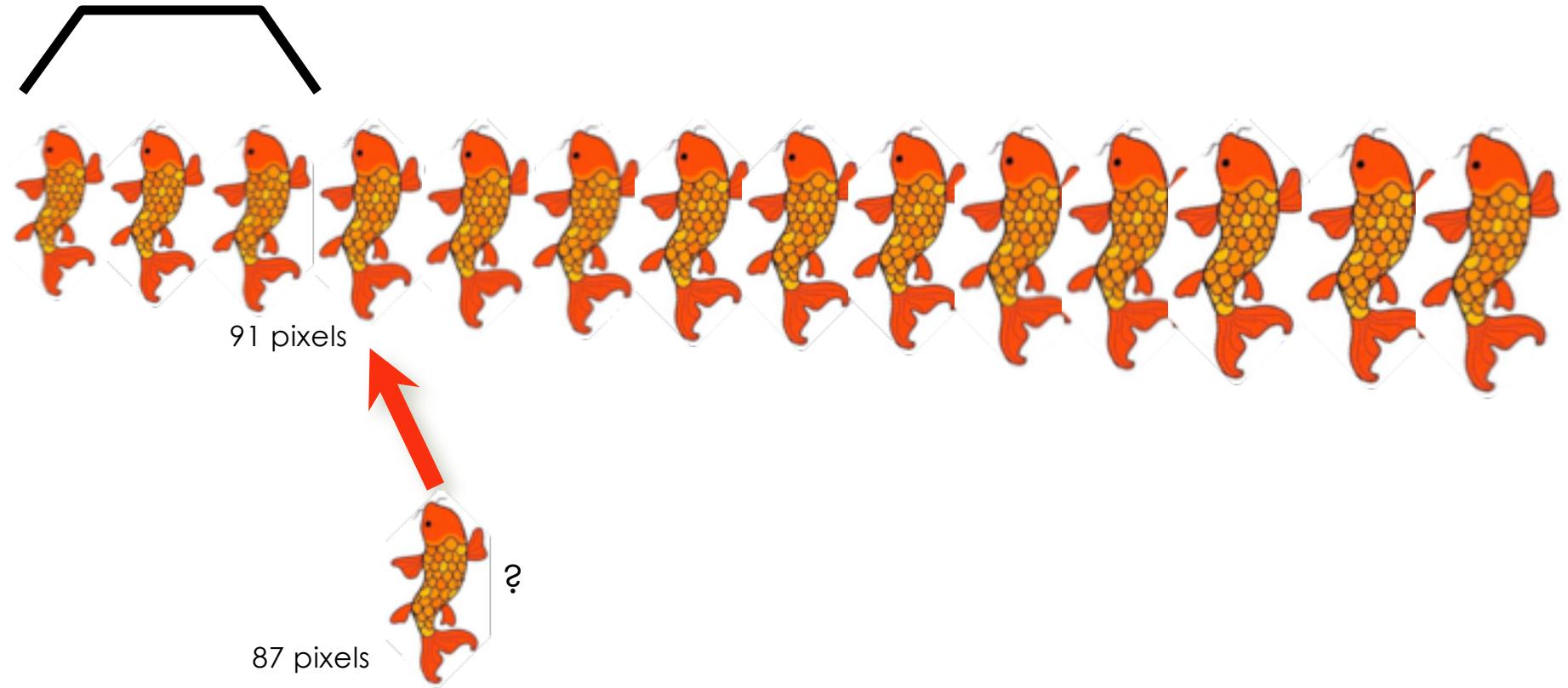
# Binary Search in an Ordered List



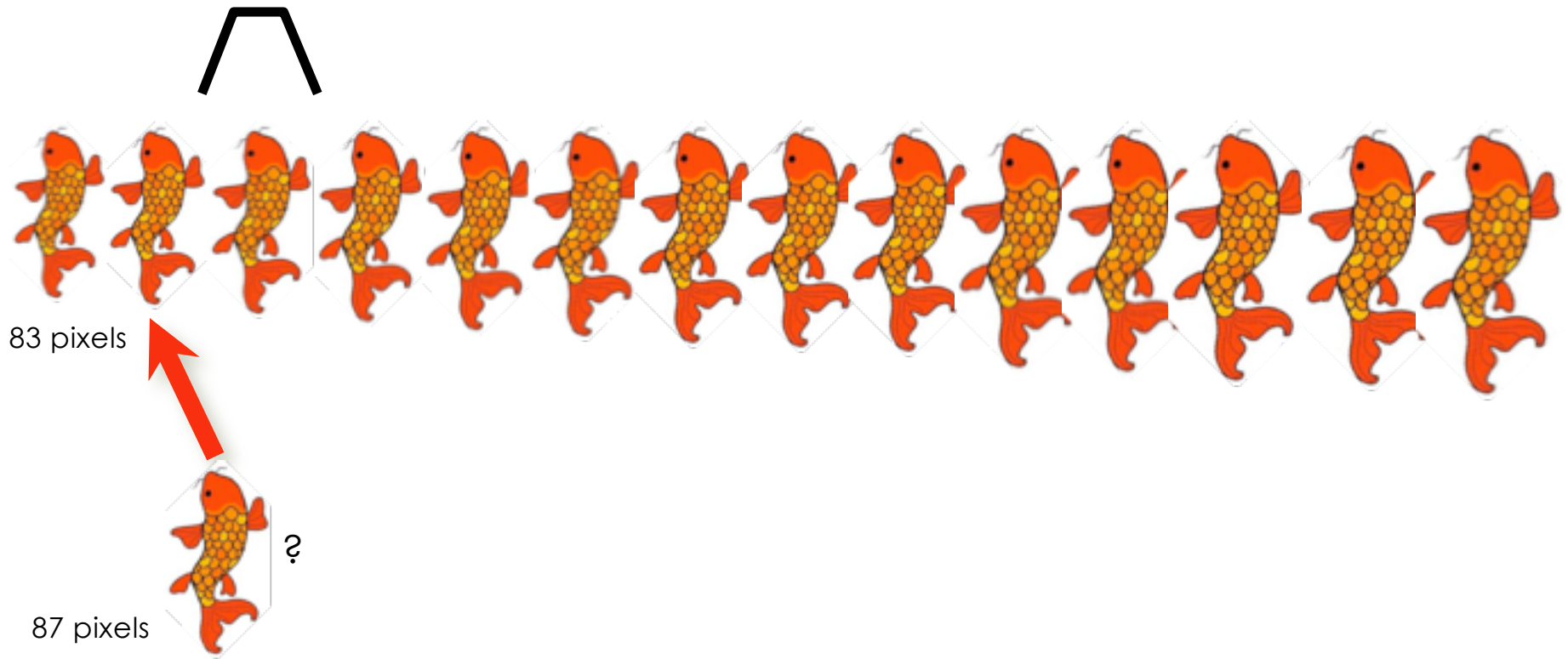
# Binary Search in an Ordered List



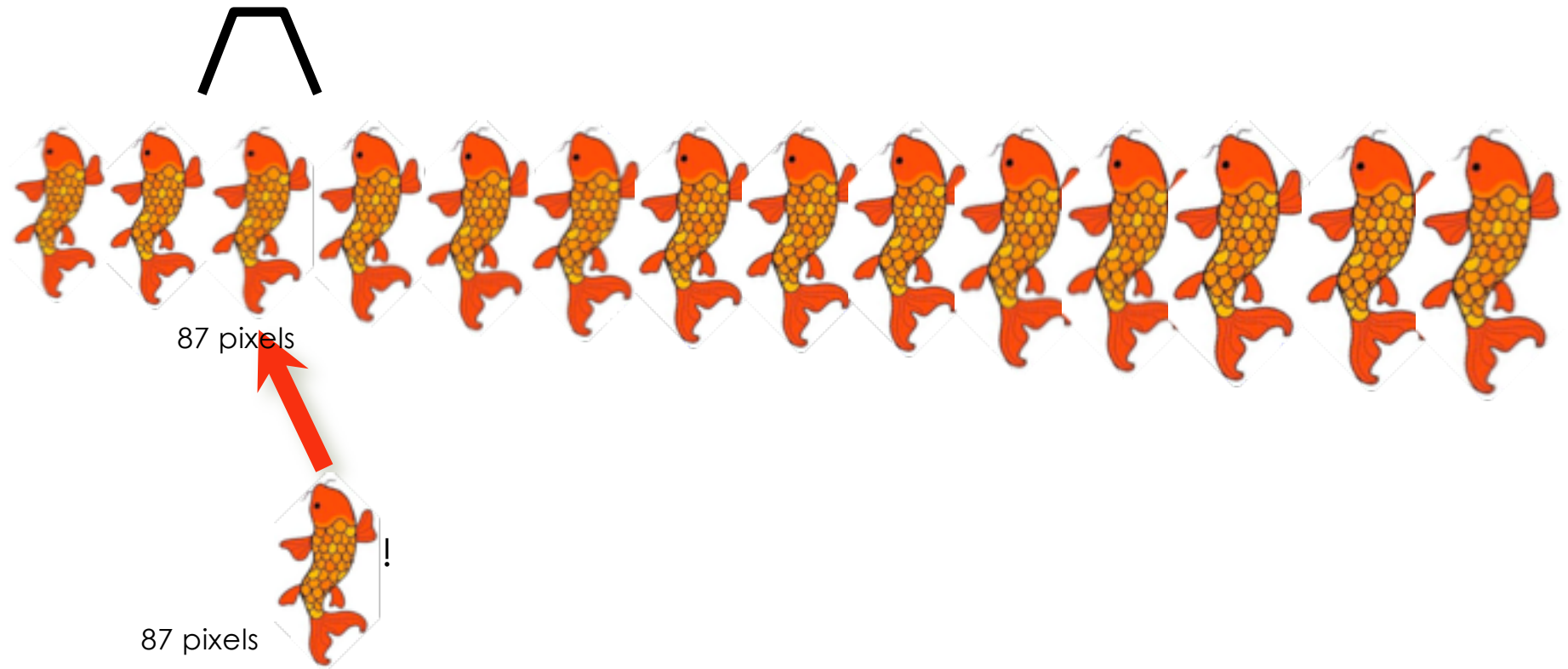
# Binary Search in an Ordered List



# Binary Search in an Ordered List



# Binary Search in an Ordered List



From idea to algorithm

# Specification: the Search Problem

- **Input:** A **list** of  $n$  unique elements and a **key** to search for
  - The elements are **sorted** in increasing order.
- **Result:** The index of an element matching the **key**, or `None` if the key is not found.



# Recursive Algorithm

`BinarySearch`(list, key):

1. Return None if the list is empty.
2. Compare the key to the middle element of the list
3. Return the index of the middle element if they match
4. If the key is less than the middle element then  
return `BinarySearch`(first half of list,key)  
Otherwise, return `BinarySearch`(second half of list,key).

# Example 1: Search for 73

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
12	25	32	37	41	48	58	60	66	73	74	79	83	91	95

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
12	25	32	37	41	48	58	60	66	73	74	79	83	91	95

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
12	25	32	37	41	48	58	60	66	73	74	79	83	91	95

Found: return 9

# Example 2: Search for 42

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14

---

12 25 32 37 41 48 58 60 66 73 74 79 83 91 95

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14

---

12 25 32 37 41 48 58 60 66 73 74 79 83 91 95

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14

---

12 25 32 37 41 48 58 60 66 73 74 79 83 91 95

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14

---

12 25 32 37 41 48 58 60 66 73 74 79 83 91 95

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14

---

12 25 32 37 41 48 58 60 66 73 74 79 83 91 95

**Not found: return None**

# Controlling the range of the search

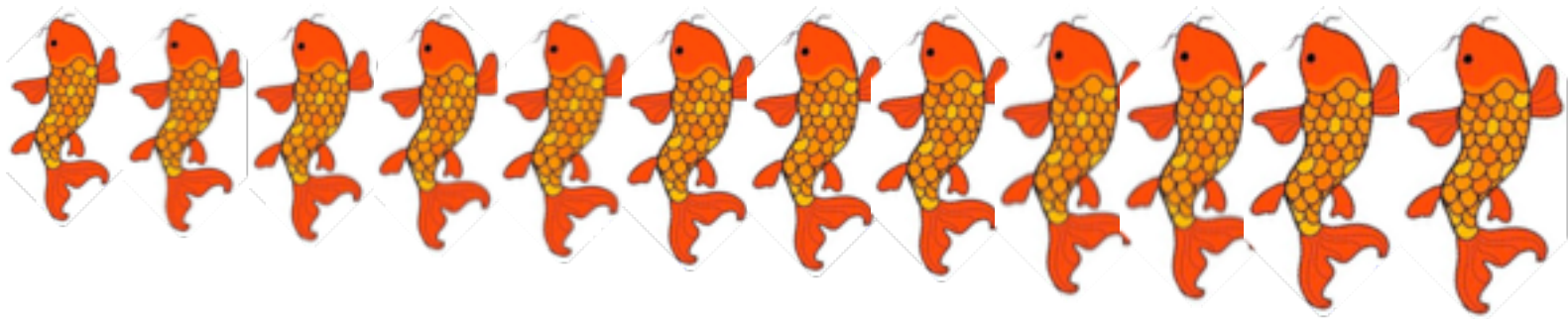
- Maintain three numbers: *lower*, *upper*, *mid*
- Initially *lower* is -1, *upper* is length of the list

*lower* = -1

*upper* =  
12

0

11



# Controlling the range of the search

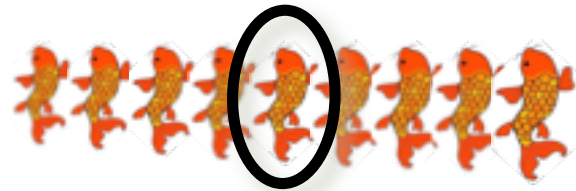
- *mid* is the midpoint of the range:

$$mid = (lower + upper) // 2 \text{ (integer division)}$$

Example: *lower* = -1, *upper* = 9

(range has 9 elements)

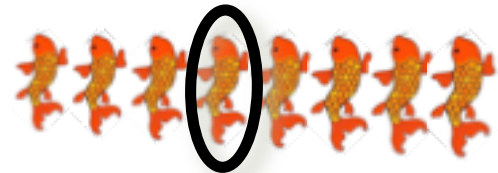
*mid* = 4



- What happens if the range has an even number of elements?

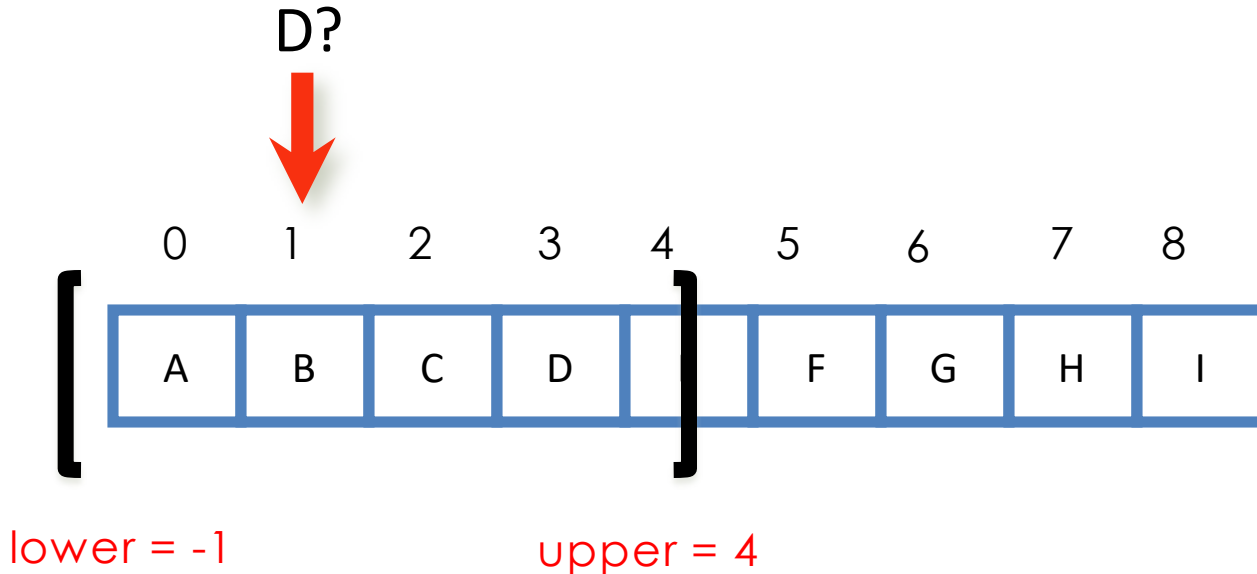
Example: *lower* = -1, *upper* = 8

*mid* = 3



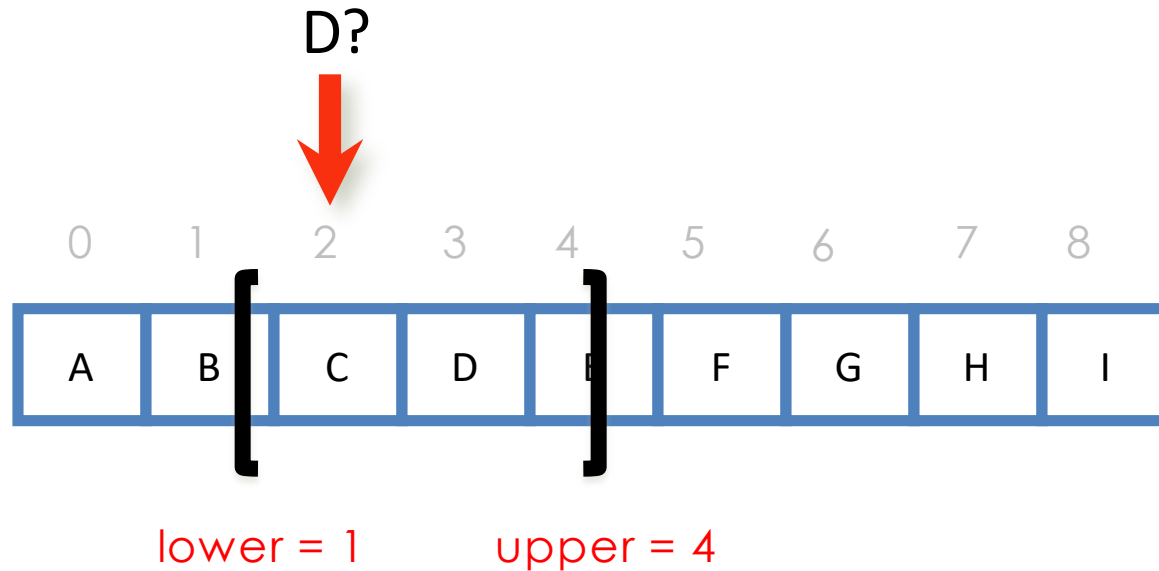


# Example



Each time we look at a smaller portion of the list within the window and ignore all the elements outside of the window

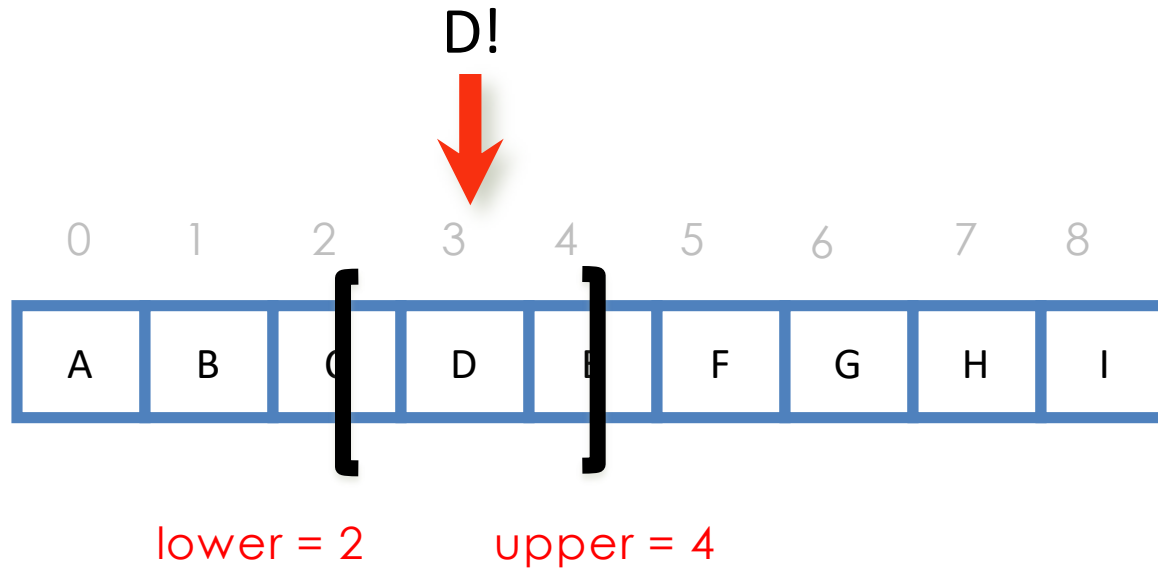
# Example



Each time we look at a smaller portion of the array within the window and ignore all the elements outside of the window



# Example



Each time we look at a smaller portion of the array within the window and ignore all the elements outside of the window

# Designing the recursion

towards a Python program:

# Base case: range empty

- How do we determine if the range is empty?
  - ▣ *lower + 1 == upper*
- What should we return then?
  - ▣ None

# Base case: key found

- The key is compared to the element at *mid*:
  - ▣  $list[mid] == key$
- What should we return then?
  - ▣ *mid*

# Recursive Case

- Non-empty range: what subproblem(s) should we solve?
  - search left half or search right half
- What should we return then?
  - result of searching left or right half
- New value for *lower*? value for *upper*?
  - left half: *lower, mid*
  - right half: *mid, upper*

# Parameters for recursion

- ▣ Inputs: *key* and list of *items*
- ▣ But we also need *lower* and *upper* bounds
  - ▣ since they change throughout the search, they have to be parameters of the search function
- ▣ Design: *main function* and *recursive helper function*

# Recursive Binary Search in Python

```
# main function
def bsearch(items, key):
    return bs_helper(items, key, -1, len(items))

# recursive helper function
def bs_helper(items, key, lower, upper):
    if lower + 1 == upper: # Base case: empty
        return None
    mid = (lower + upper) // 2 # Recursive case
    if key == items[mid]:
        return mid
    if key < items[mid]: # Go left
        return bs_helper(items, key, lower, mid)
    else: # Go right
        return bs_helper(items, key, mid, upper)
```

Diagram illustrating the recursive binary search function with parameter updates:

- Initial call:** `bsearch(items, key)` calls `bs_helper(items, key, -1, len(items))`.
  - `-1` is the **first value for lower**.
  - `len(items)` is the **first value for upper**.
- Recursive case (Left branch):** `bs_helper(items, key, lower, mid)`.
  - `lower` is the **same value for lower**.
  - `mid` is the **new value for upper**.
- Recursive case (Right branch):** `bs_helper(items, key, mid, upper)`.
  - `mid` is the **new value for lower**.
  - `upper` is the **same value for upper**.

# Caveat: specification

- The algorithm and the code was developed on the assumption that the input list is **sorted**.
- If the function is called with an unsorted list it has no obligation to behave correctly.



# measurement and analysis

reflections

# Trace: Search for 73

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14

12 25 32 37 41 48 58 60 66 73 74 79 83 91 95

key lower upper

```
bs_helper(items, 73, -1, 15)
```

```
mid = 7 and 73 > items[7]
```

```
bs_helper(items, 73, 7, 15)
```

```
mid = 11 and 73 < items[11]
```

```
bs_helper(items, 73, 7, 11)
```

```
mid = 9 and 73 == items[9]
```

```
----> return 9
```

# Trace: Search for 42

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14  
12 25 32 37 41 48 58 60 66 73 74 79 83 91 95

key lower upper

```
bs_helper(items, 42, -1, 15)
```

```
mid = 7 and 42 < items[7]
```

```
bs_helper(items, 42, -1, 7)
```

```
mid = 3 and 42 > items[3]
```

```
bs_helper(items, 42, 3, 7)
```

```
mid = 5 and 42 < items[5]
```

```
bs_helper(items, 42, 3, 5)
```

```
mid = 4 and 42 > items[4]
```

```
bs_helper(items, 73, 4, 5)
```

```
lower + 1 == upper
```

```
----> Return None.
```

# Instrumenting Binary Search Code

```
count = 0 # count of comparisons

def bsearch(list, key):
    global count
    count = 0
    print("Searching list of length ", len(list))
    result = bs_helper(list, key, -1, len(list))
    print("Number of comparisons:", count)
    return result

def bs_helper(list, key, lower, upper):
    global count
    if lower + 1 == upper:
        print("Not found")
        return None
    mid = (lower + upper) // 2
    print("mid:", mid, "lower:", lower, "upper", upper)
    count = count + 1
    if key == list[mid]:
        return mid
    if key < list[mid]:
        return bs_helper(list, key, lower, mid)
    else:
        return bs_helper(list, key, mid, upper)
```

# Instrumented Output

```
>>> bsearch(list(range(1,500,2)), 21)
```

```
Searching list of length 250  
mid: 124 lower: -1 upper 250  
mid: 61 lower: -1 upper 124  
mid: 30 lower: -1 upper 61  
mid: 14 lower: -1 upper 30  
mid: 6 lower: -1 upper 14  
mid: 10 lower: 6 upper 14  
Number of comparisons: 6  
10
```

```
>>> bsearch(list(range(1,500,2)), 256)
```

```
Searching list of length 250  
mid: 124 lower: -1 upper 250  
mid: 187 lower: 124 upper 250  
mid: 155 lower: 124 upper 187  
mid: 139 lower: 124 upper 155  
mid: 131 lower: 124 upper 139  
mid: 127 lower: 124 upper 131  
mid: 129 lower: 127 upper 131  
mid: 128 lower: 127 upper 129  
Not found  
Number of comparisons: 8
```

```
>>> bsearch(list(range(1,500000,2)), 256)
```

```
Searching list of length 250000  
mid: 124999 lower: -1 upper 250000  
...  
mid: 127 lower: 126 upper 128  
Not found  
Number of comparisons: 18
```

```
>>> bsearch(list(range(1,5000000,2)), 256)
```

```
Searching list of length 2500000  
mid: 1249999 lower: -1 upper 2500000  
...  
mid: 128 lower: 127 upper 129  
Not found  
Number of comparisons: 21
```

# Analyzing Binary Search

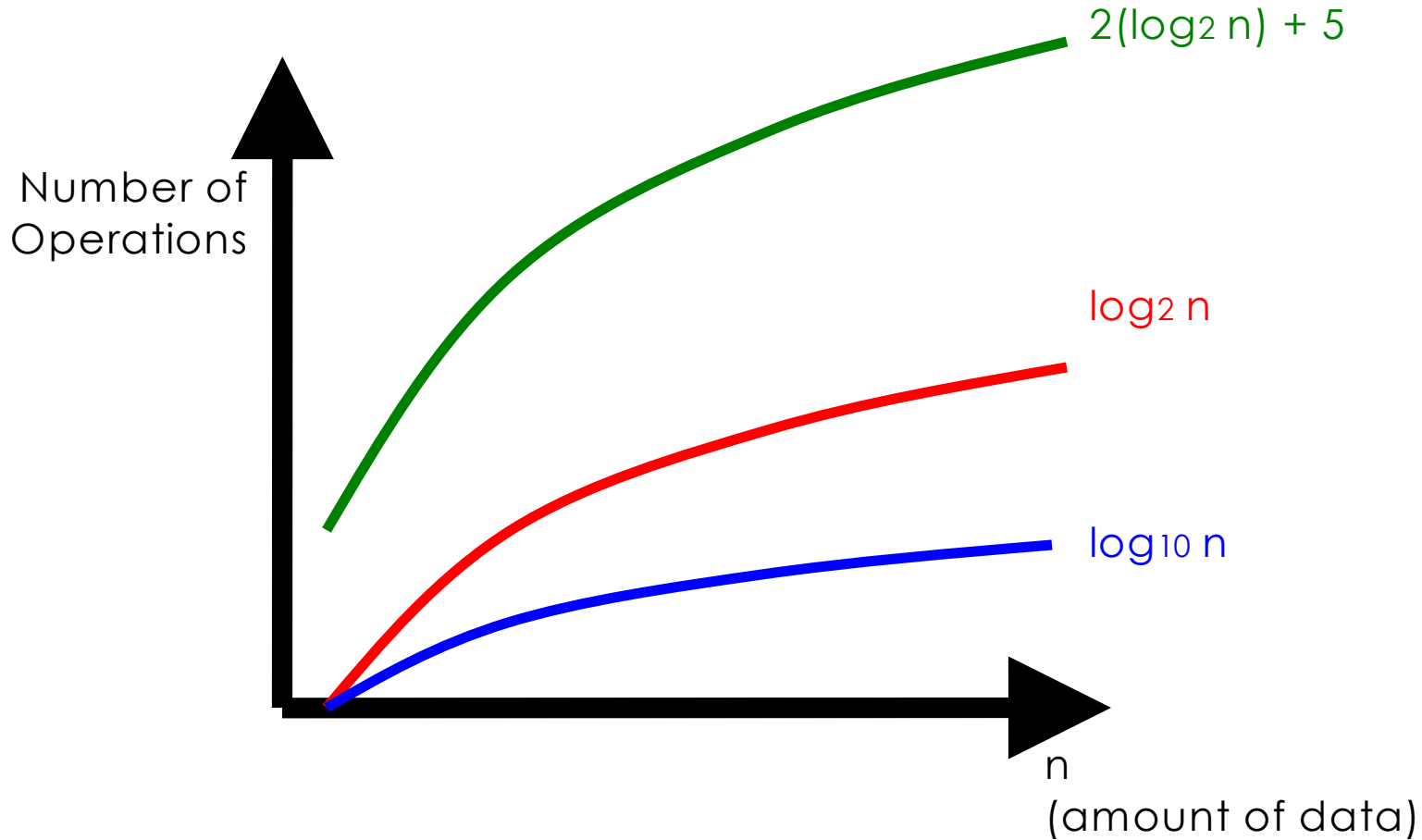
- ▣ Suppose we search for a key larger than anything in the list.
- ▣ Example sequences of range sizes:
  - 8, 4, 2, 1 (4 key comparisons)
  - 16, 8, 4, 2, 1 (5 key comparisons)
  - 17, 8, 4, 2, 1 (5 key comparisons)
  - 18, 9, 4, 2, 1 (5 key comparisons)
  
  - 31, 15, 7, 3, 1 (still 5 key comparisons)
  - 32, 16, 8, 4, 2, 1 (at last, 6 key comparisons)
- ▣ Notice:  $8 = 2^3$ ,  $16 = 2^4$ ,  $32 = 2^5$
- ▣ Therefore:  $\log 8 = 3$ ,  $\log 16 = 4$ ,  $\log 32 = 5$

# Generalizing the Analysis

“floor”

- Some notation:  $\lfloor x \rfloor$  means round  $x$  down, so  $\lfloor 2.5 \rfloor = 2$
- Binary search of  $n$  elements will do at most  $1 + \lfloor \log_2 n \rfloor$  comparisons  
 $1 + \lfloor \log_2 8 \rfloor = 1 + \lfloor \log_2 9 \rfloor = \dots 1 + \lfloor \log_2 15 \rfloor = 4$   
 $1 + \lfloor \log_2 16 \rfloor = 1 + \lfloor \log_2 17 \rfloor = \dots 1 + \lfloor \log_2 31 \rfloor = 5$
- Why? We can split search region in half  $1 + \lfloor \log_2 n \rfloor$  times before it becomes empty.
- "Big O" notation: we ignore the "1 +" and the floor function. **We say Binary Search has complexity  $O(\log n)$ .**

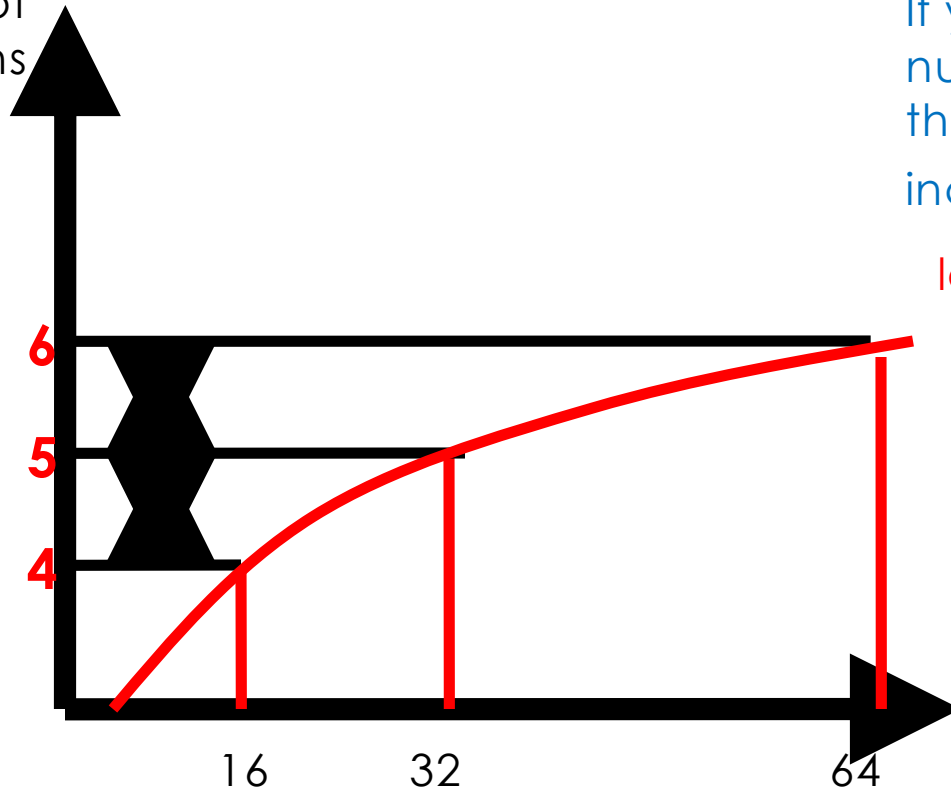
# $O(\log n)$ ("logarithmic time")





# $O(\log n)$

Number of Operations



For a  $\log_2 n$  algorithm,  
If you double the  
number of data elements  
the amount of work you do  
increases by just one unit

$\log_2 n$

n  
(amount of data)

# Binary Search (Worst Case)

Number of elements

Number of Comparisons

15

4

31

5

63

6

127

7

255

8

511

9

1023

10

1 million

20

# Binary Search Pays Off

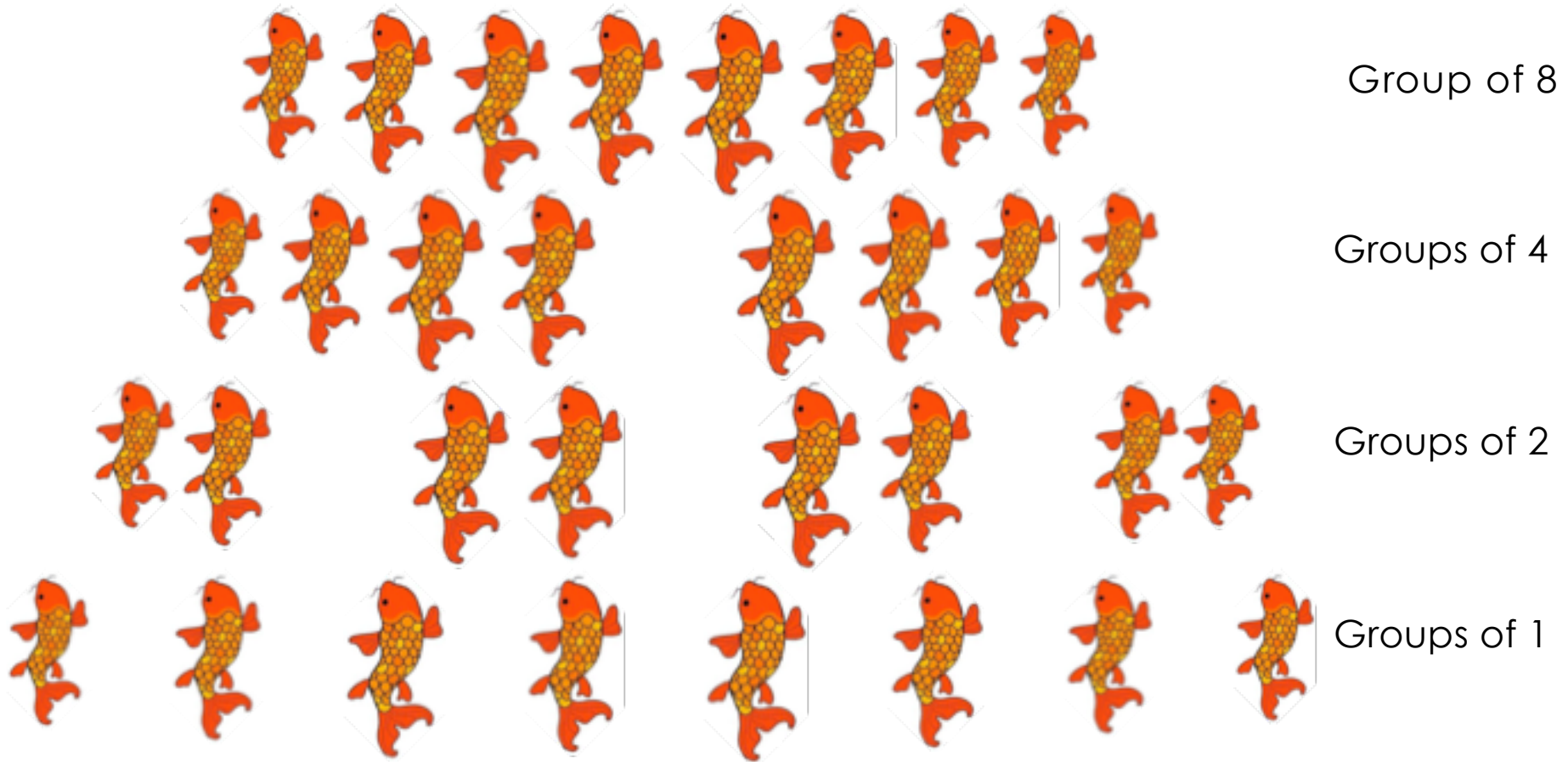
- Finding an element in an list with a million elements requires only 20 comparisons!
  - BUT....
    - The list must be sorted.
    - What if we sort the list first using insertion sort?
      - Insertion sort  $O(n^2)$  (worst case)
      - Binary search  $O(\log n)$  (worst case)
      - Total time:  $O(n^2) + O(\log n) = O(n^2)$
- Luckily there are faster ways to sort in the worst case...

# Merge Sort

# Divide and Conquer

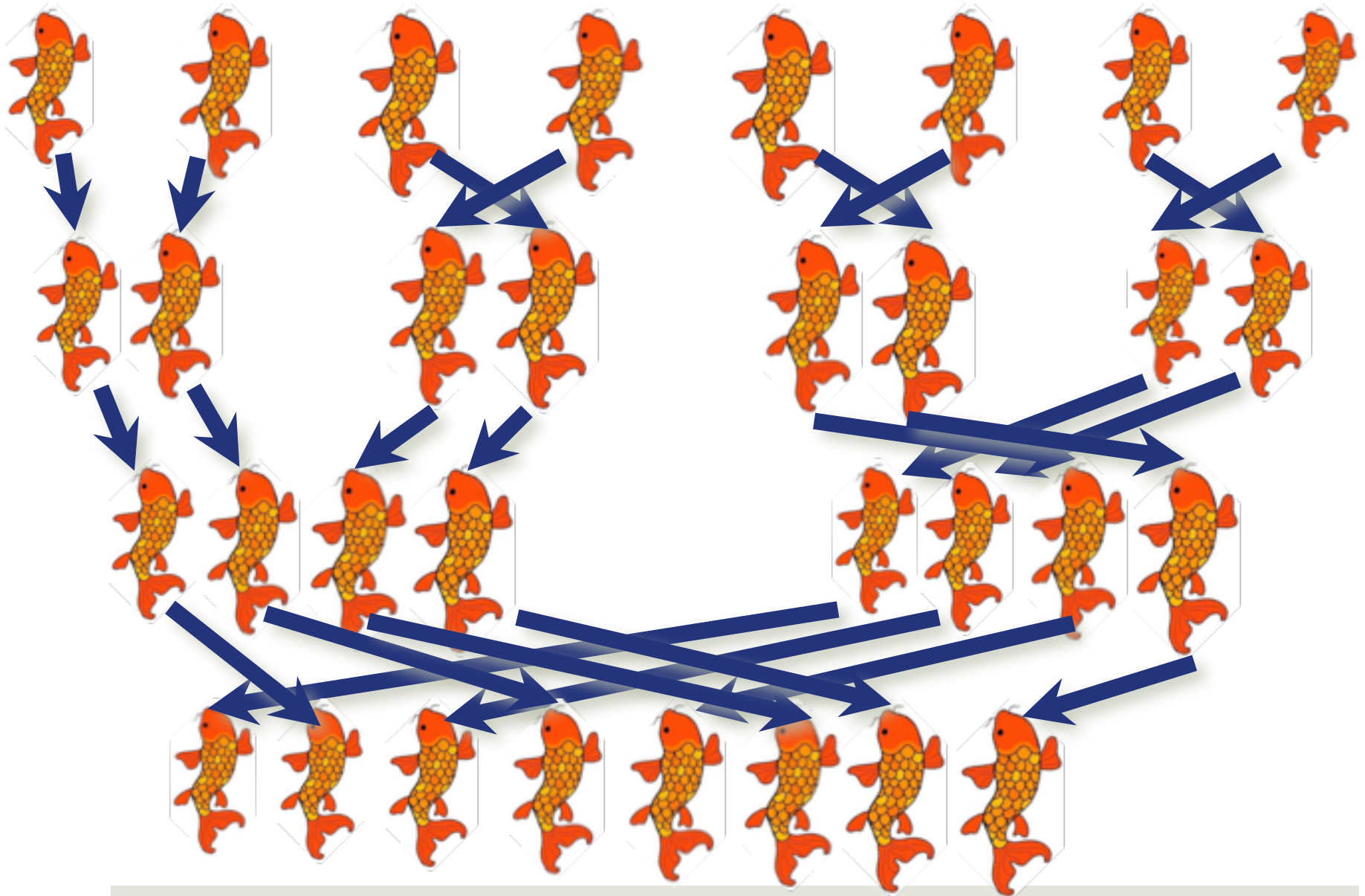
- In computation:
  - **Divide** the problem into “simpler” versions of itself.
  - **Conquer** each problem using the same process (usually recursively).
  - **Combine** the results of the “simpler” versions to form your final solution.
- Examples: Towers of Hanoi, fractals, Binary Search, Merge Sort, Quicksort, and many, many more

# Divide

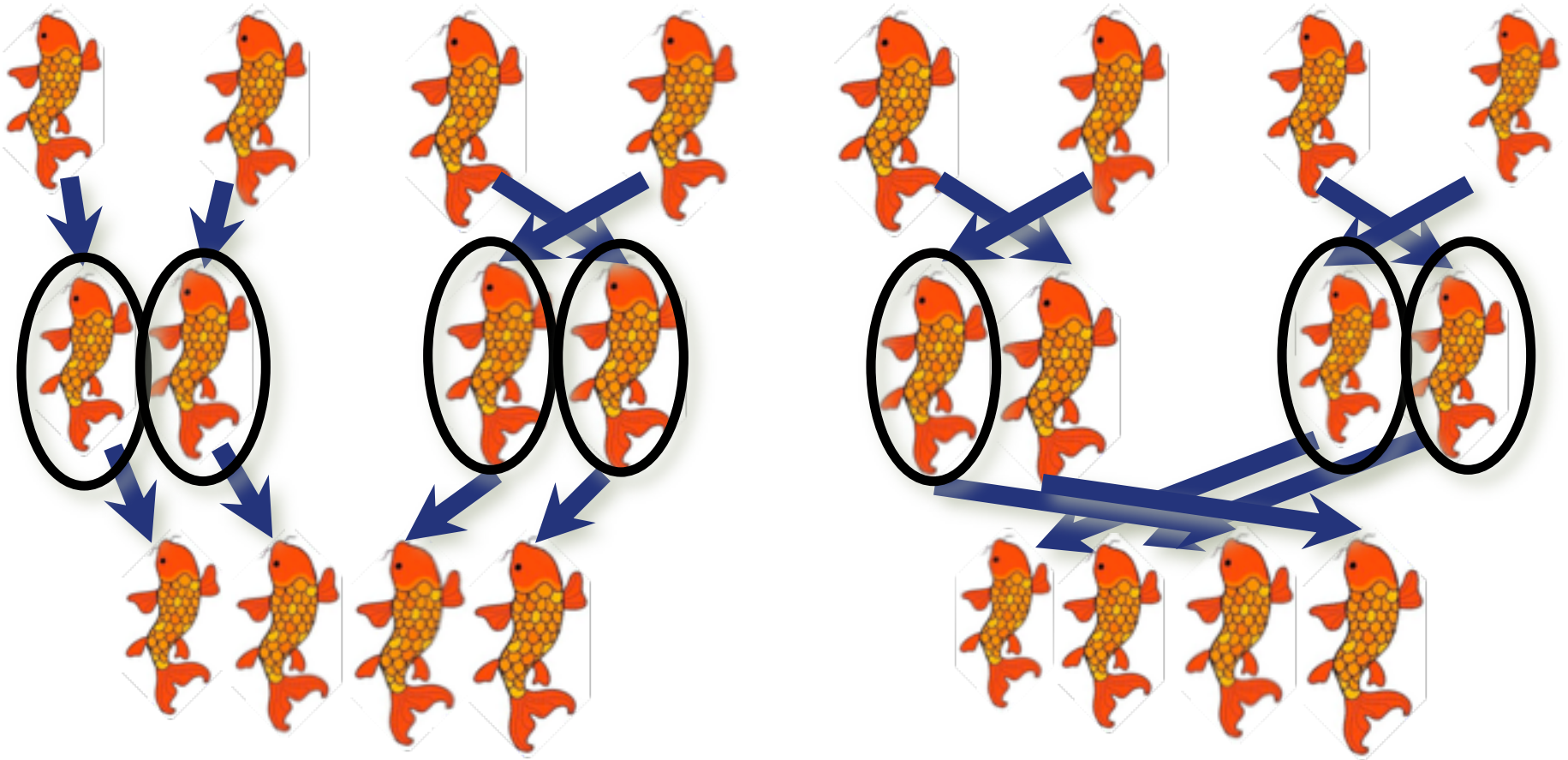


**Now each "group" is (trivially) sorted!**

# Conquer (merge sorted lists)

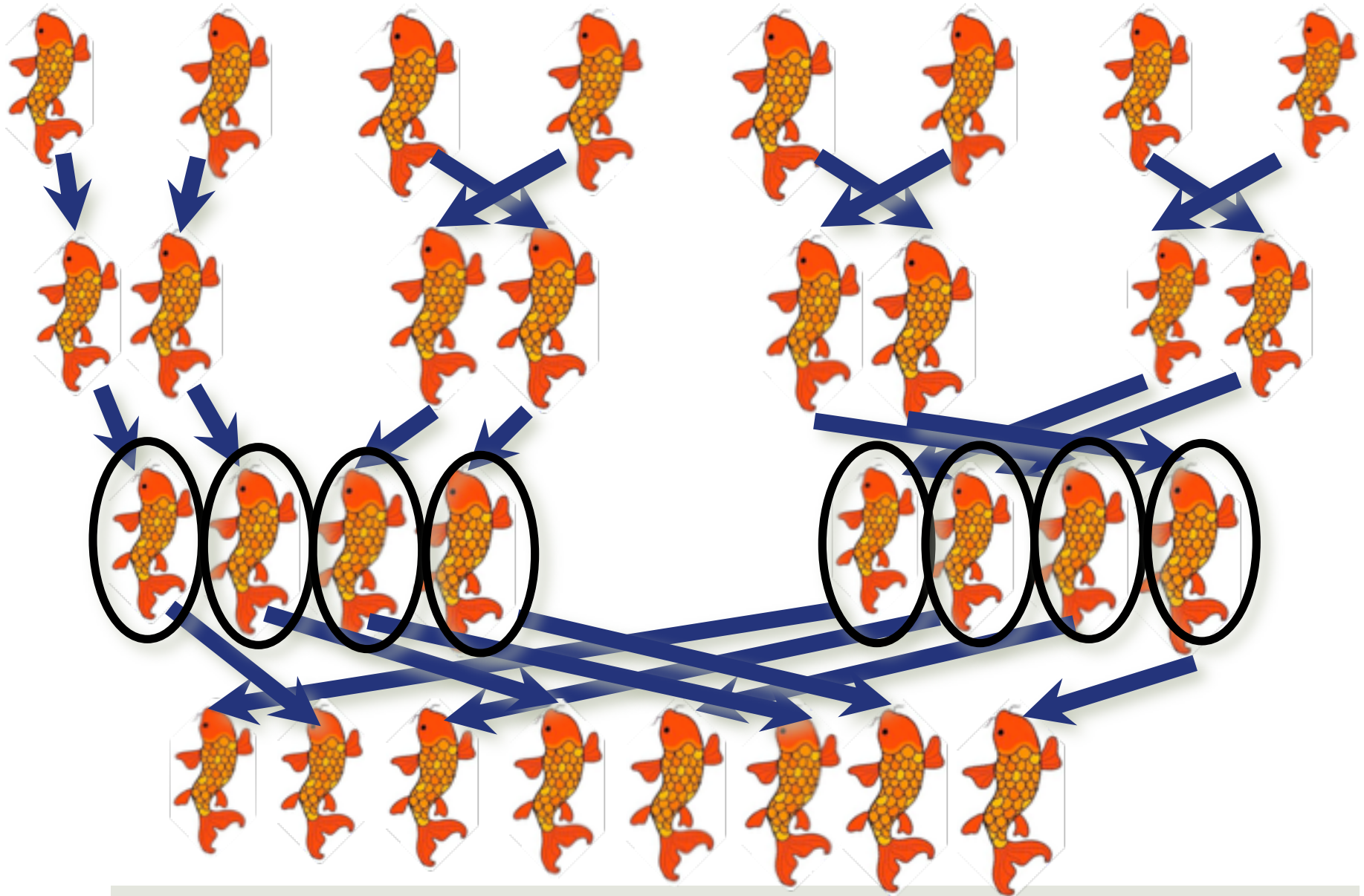


# Conquer (merge sorted lists)





# Conquer (merge sorted lists)



# Merge Sort

- **Input:** List  $a$  of  $n$  elements.
- **Output:** Returns **a new list** containing the same elements in sorted order.
- **Algorithm:**
  1. If less than two elements, return a **copy** of the list (**base case!**)
  2. Sort the first half using merge sort. (**recursive!**)
  3. Sort the second half using merge sort. (**recursive!**)
  4. **Merge** the two sorted halves to obtain the final sorted array.

# Merge Sort in Python

```
def msort(list):
    if len(list) == 0 or len(list) == 1: # base case
        return list[:len(list)] # copy the input

    # recursive case
    halfway = len(list) // 2
    list1 = list[0:halfway]
    list2 = list[halfway:len(list)]
    newlist1 = msort(list1) # recursively sort left half
    newlist2 = msort(list2) # recursively sort right half
    newlist = merge(newlist1, newlist2)
    return newlist
```

# Merge Outline

- **Input:** Two lists  $a$  and  $b$ , **already sorted**
- **Output:** A new list containing the elements of  $a$  and  $b$  merged together in sorted order.
- **Algorithm:**
  1. Create an empty list  $c$ , set  $index\_a$  and  $index\_b$  to 0
  2. While  $index\_a < \text{length of } a$  and  $index\_b < \text{length of } b$ 
    - a. Add the smaller of  $a[index\_a]$  and  $b[index\_b]$  to the end of  $c$ , and increment the index of the list with the smaller element
  3. If any elements are left over in  $a$  or  $b$ , add them to the end of  $c$ , in order
  4. Return  $c$

# Filling in the details of Merge

"Add the smaller of  $a[index\_a]$  and  $b[index\_b]$  to the end of  $c$ , and increment the index of the list with the smaller element":

a. If  $a[index\_a] \leq b[index\_b]$ , then do the following:

- i. append  $a[index\_a]$  to the end of  $c$
- ii. add 1 to  $index\_a$

b. Otherwise, do the following:

- i. append  $b[index\_b]$  to the end of  $c$
- ii. add 1 to  $index\_b$

# Filling in the details of Merge

"If any elements are left over in  $a$  or  $b$ , add them to the end of  $c$ , in order":

a. If  $index\_a <$  the length of list  $a$ , then:

- i. append all remaining elements of list  $a$  to the end of list  $c$ , in order

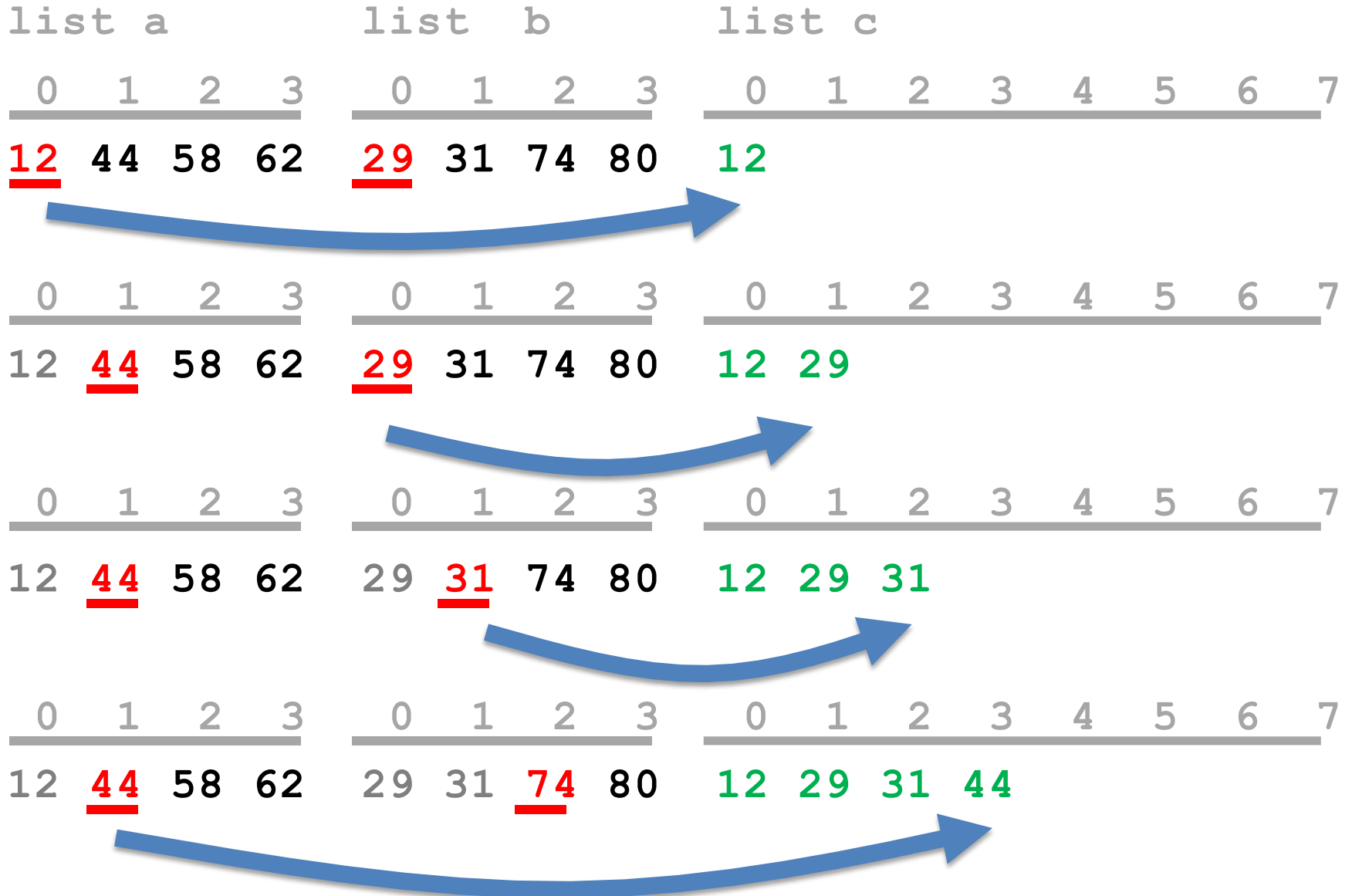
b. Otherwise:

- i. append all remaining elements of list  $b$  (if any) to the end of list  $c$ , in order

# Merge in Python

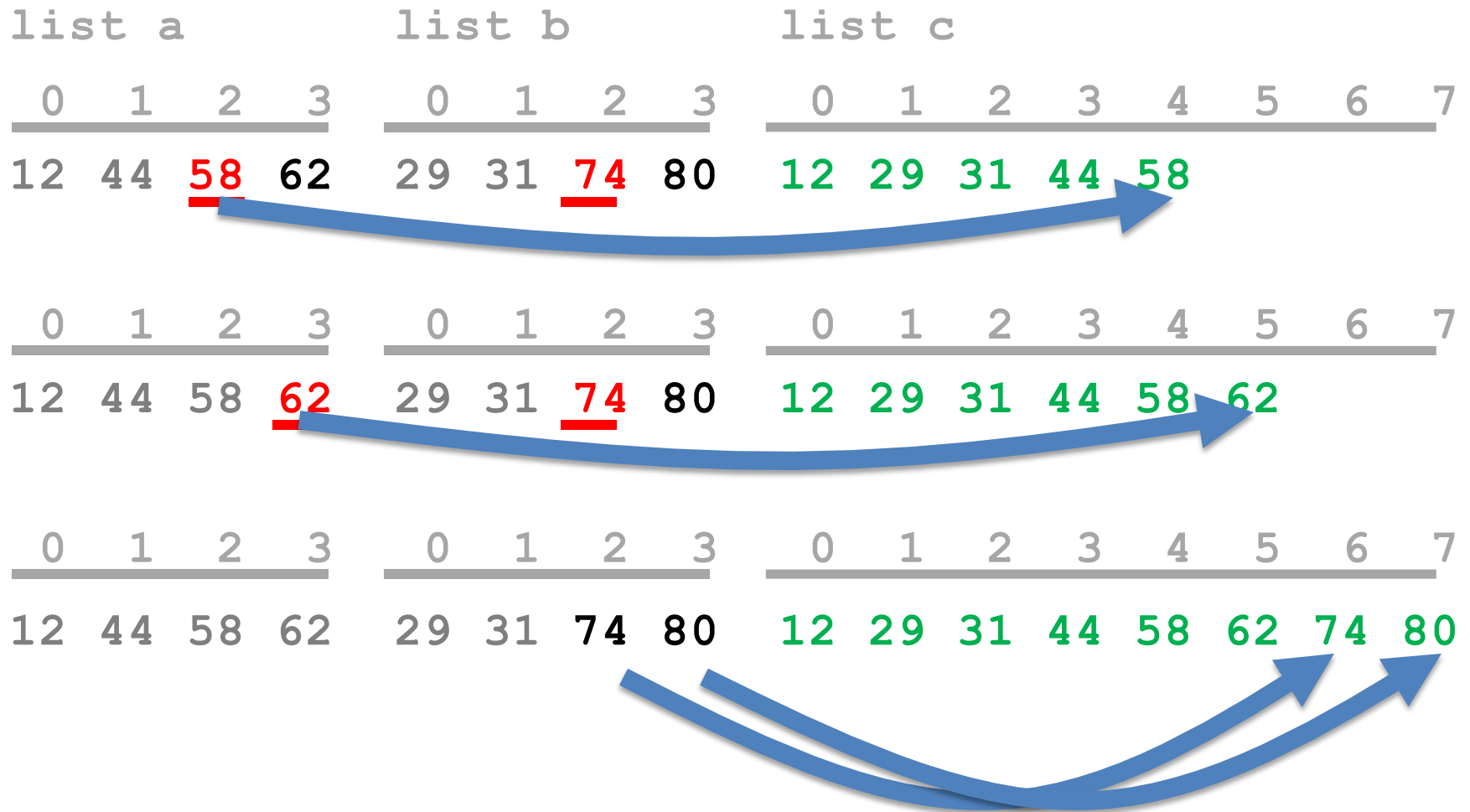
```
def merge(a, b):
    index_a = 0
    index_b = 0
    c = []
    while index_a < len(a) and index_b < len(b):
        if a[index_a] <= b[index_b]:
            c.append(a[index_a])
            index_a = index_a + 1
        else:
            c.append(b[index_b])
            index_b = index_b + 1
    # when we exit the loop
    # we are at the end of at least one of the lists
    c.extend(a[index_a:])    c.extend(b[index_b:])
    return c
```

# Example 1: Merge





# Example 1: Merge (cont'd)



# Example 2: Merge

list a

0	1	2	3
<u>58</u>	67	74	90

0	1	2	3
<u>58</u>	67	74	90

0	1	2	3
<u>58</u>	67	74	90

0	1	2	3
<u>58</u>	67	74	90

0	1	2	3
58	67	74	90

list b

0	1	2	3
<u>19</u>	26	31	44

0	1	2	3
19	<u>26</u>	31	44

0	1	2	3
19	26	<u>31</u>	44

0	1	2	3
19	26	31	<u>44</u>

0	1	2	3
19	26	31	44

list c

0	1	2	3	4	5	6	7
19							

0	1	2	3	4	5	6	7
19	26						

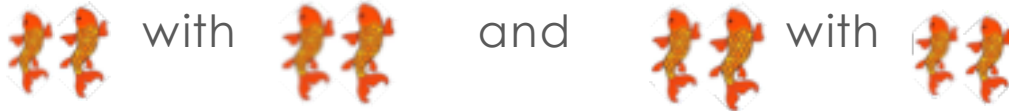
0	1	2	3	4	5	6	7
19	26	31					

0	1	2	3	4	5	6	7
19	26	31	44				

0	1	2	3	4	5	6	7
19	26	31	44	58	67	74	90

# Analyzing Efficiency

- **Constant time** operations: comparing values and appending elements to the output.
- If you merge two lists of size  $i/2$  into one new list of size  $i$ , what is the **maximum number of appends** that you must do? **maximum number of comparisons?**
- Example: say we are merging two pairs of 2-element lists:




8 appends for 8 elements

- **If you have a group of lists to be merged pairwise, and the total number of elements in the whole group is  $n$ , the total number of appends will be  $n$ .**
- **Worse case number comparisons?  $n/2$  or less, but still  $O(n)$**

# How many merges?

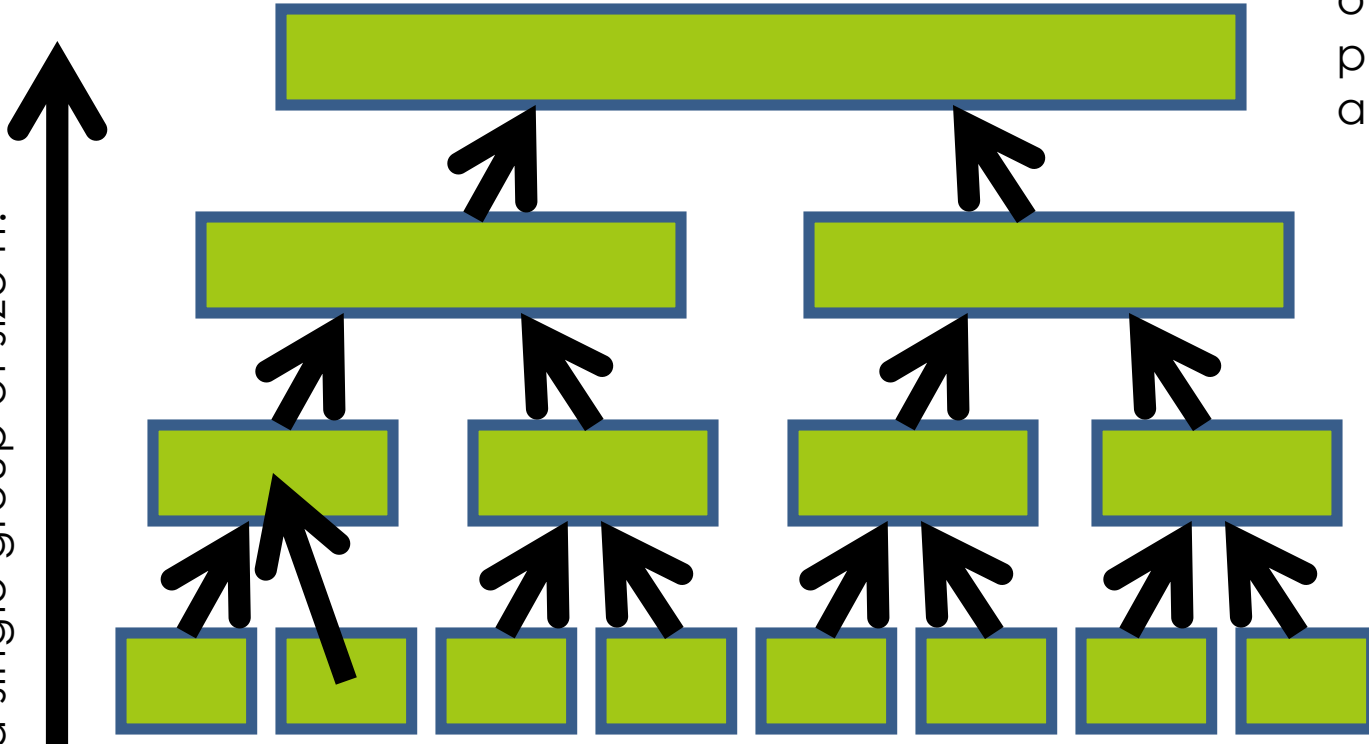
- We saw that each group of merges of  $n$  elements takes  $O(n)$  operations.
- How many times do we have to merge  $n$  elements to go from  $n$  groups of size 1 to 1 group of size  $n$ ?
- Example: Merge sort on 32 elements.
  - Break down to groups of size 1 (base case).
  - Merge 32 lists of size 1 into 16 lists of size 2.
  - Merge 16 lists of size 2 into 8 lists of size 4.
  - Merge 8 lists of size 4 into 4 lists of size 8.
  - Merge 4 lists of size 8 into 2 lists of size 16.
  - Merge 2 lists of size 16 into 1 list of size 32.
- **In general:  $\log_2 n$  merges of  $n$  elements.**



$5 = \log_2 32$

# Putting it all together

It takes  $\log_2 n$  merges to go from  $n$  groups of size 1 to a single group of size  $n$ .



Total number of elements per level is always  $n$ .

It takes  $n$  appends to merge all pairs to the next higher level.

**Multiply the number of levels by the number of appends per level.**

# Big O

- In the worst case, merge sort requires  **$O(n \log_2 n)$**  time to sort an array with  $n$  elements.

## Number of operations

$$n \log_2 n$$

$$(n + n/2) \log_2 n$$

$$4n \log_{10} n$$

$$n \log_2 n + 2n$$

## Order of Complexity

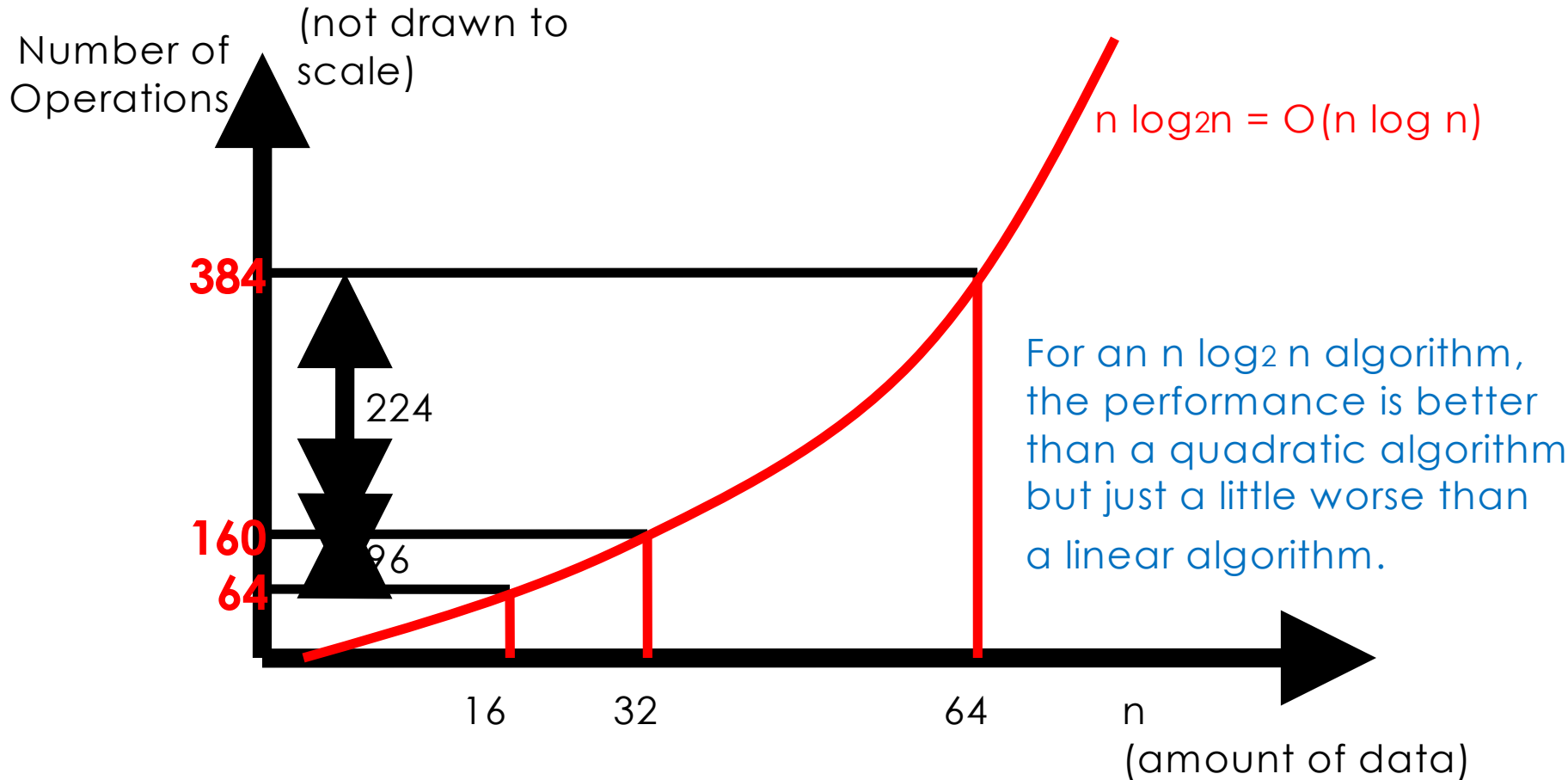
$$O(n \log n)$$

$$O(n \log n)$$

$$O(n \log n)$$

$$O(n \log n)$$

# $O(N \log N)$



# Merge vs. Insertion Sort

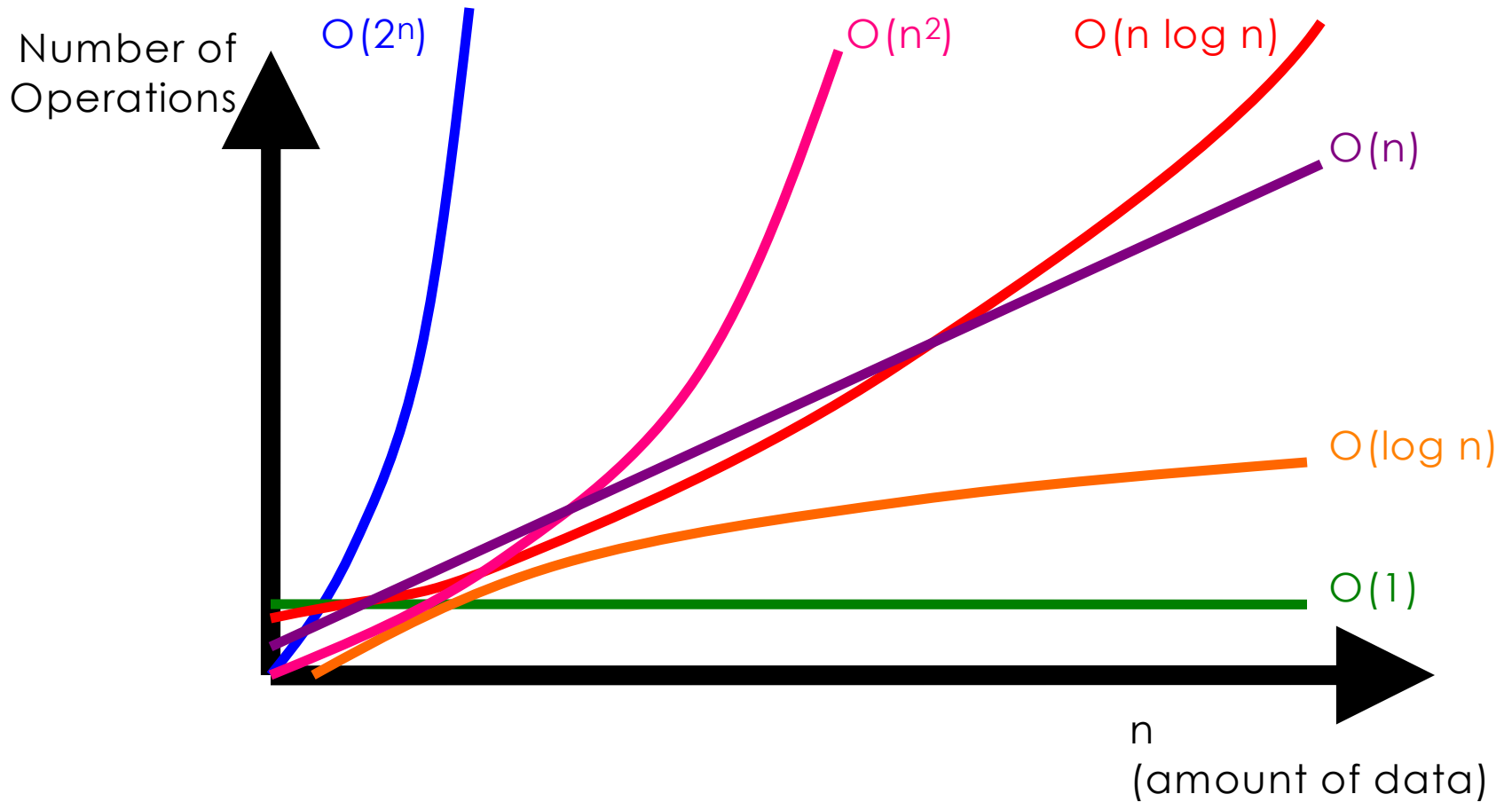
n	isort $(n(n+1)/2)$	msort $(n \log_2 n)$	Ratio
8	36	24	0.67
16	136	64	0.47
32	528	160	0.3
$2^{10}$	524,800	10,240	0.02
$2^{20}$	549,756,338,176	20,971,520	0.00004



# Sorting and Searching

- Recall that if we wanted to use binary search, the list must be sorted.
- ▣ What if we sort the array first using merge sort?
  - ▣ Merge sort  $O(n \log n)$  (worst case)
  - ▣ Binary search  $O(\log n)$  (worst case)
  - ▣ Total time:  
(worst case)  $O(n \log n) + O(\log n) = \mathbf{O(n \log n)}$

# Comparing Big O Functions



# Merge Sort: Iteratively

(optional)

- If you are interested, Explorations of Computing discusses an iterative version of merge sort which you can read on your own.
- This version uses an alternate version of the `merge` function that is not shown in the textbook but is given in PythonLabs.

# Built-in Sort in Python

- Why we study sorting algorithms
  - Practice in algorithmic thinking
  - Practice in complexity analysis
- You will **rarely** need to implement your own sort function
  - Python method `list.sort` takes a lists and modifies it while it sorts
  - Python function `sorted` takes a list and returns a new sorted list
  - Python uses *timsort* by Tim Peters (fancy!)

# Quicksort

- Conceptually similar to merge sort
- Uses the technique of divide-and-conquer
  1. Pick a pivot
  2. Divide the array into two subarrays, those that are smaller and those that are greater
  3. Put the pivot in the middle, between the two sorted arrays
- Worst case  $O(n^2)$
- "Expected"  $O(n \log n)$