UNIT 5C
Merge Sort

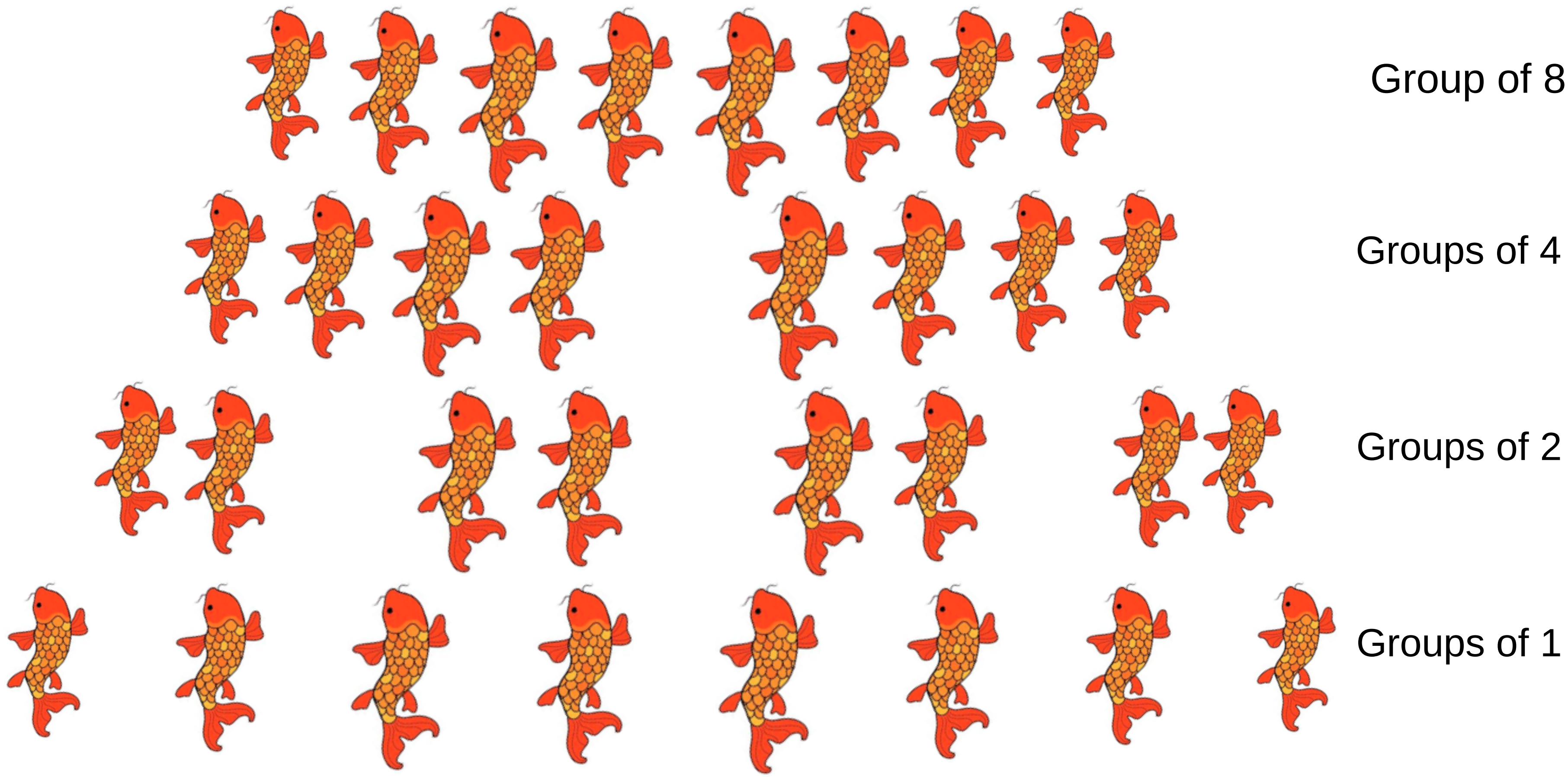
Divide and Conquer

- 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:

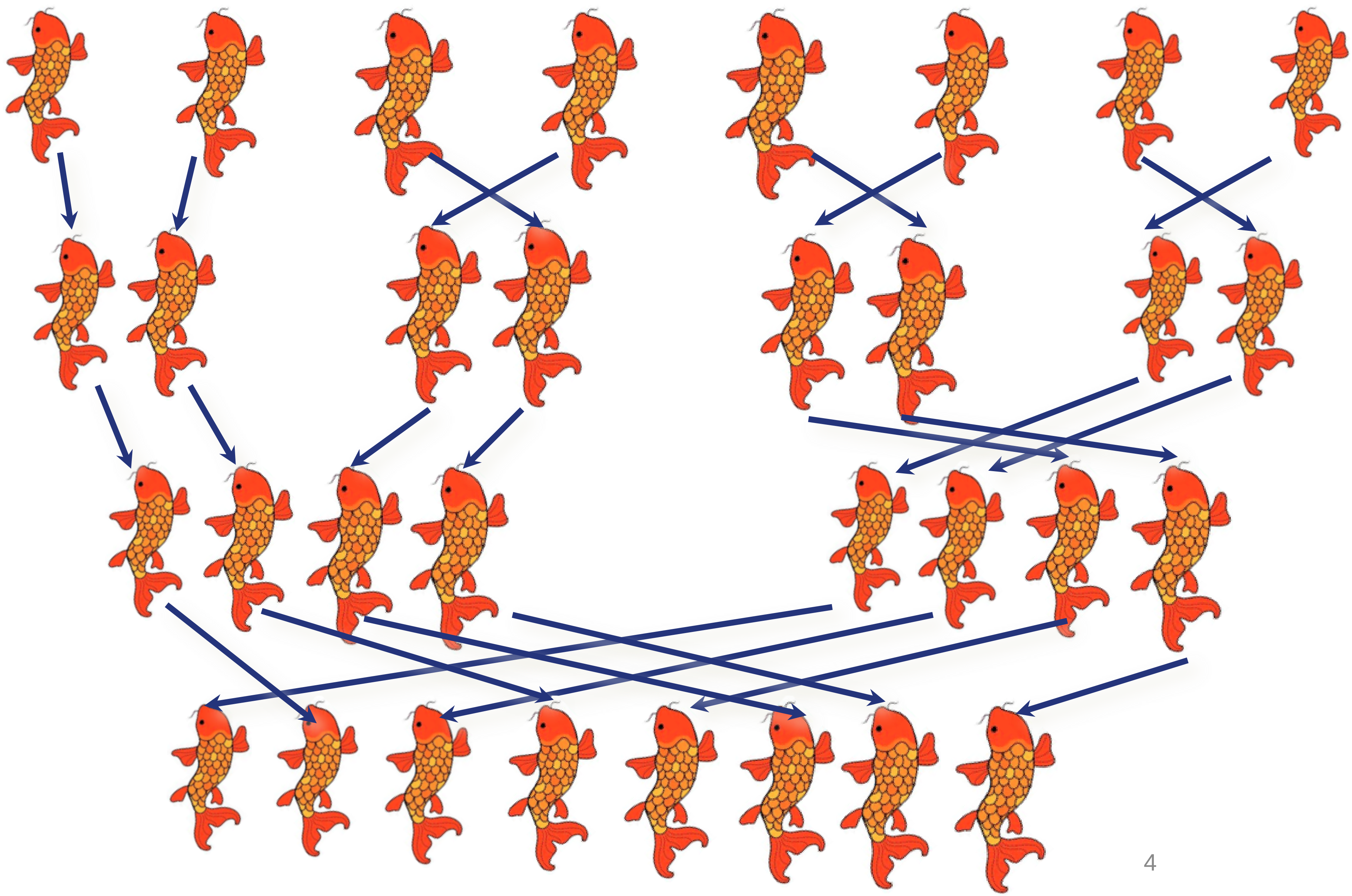
Towers of Hanoi, Fractals,
Binary Search, Merge Sort,
Quicksort,
and many, many more

Divide

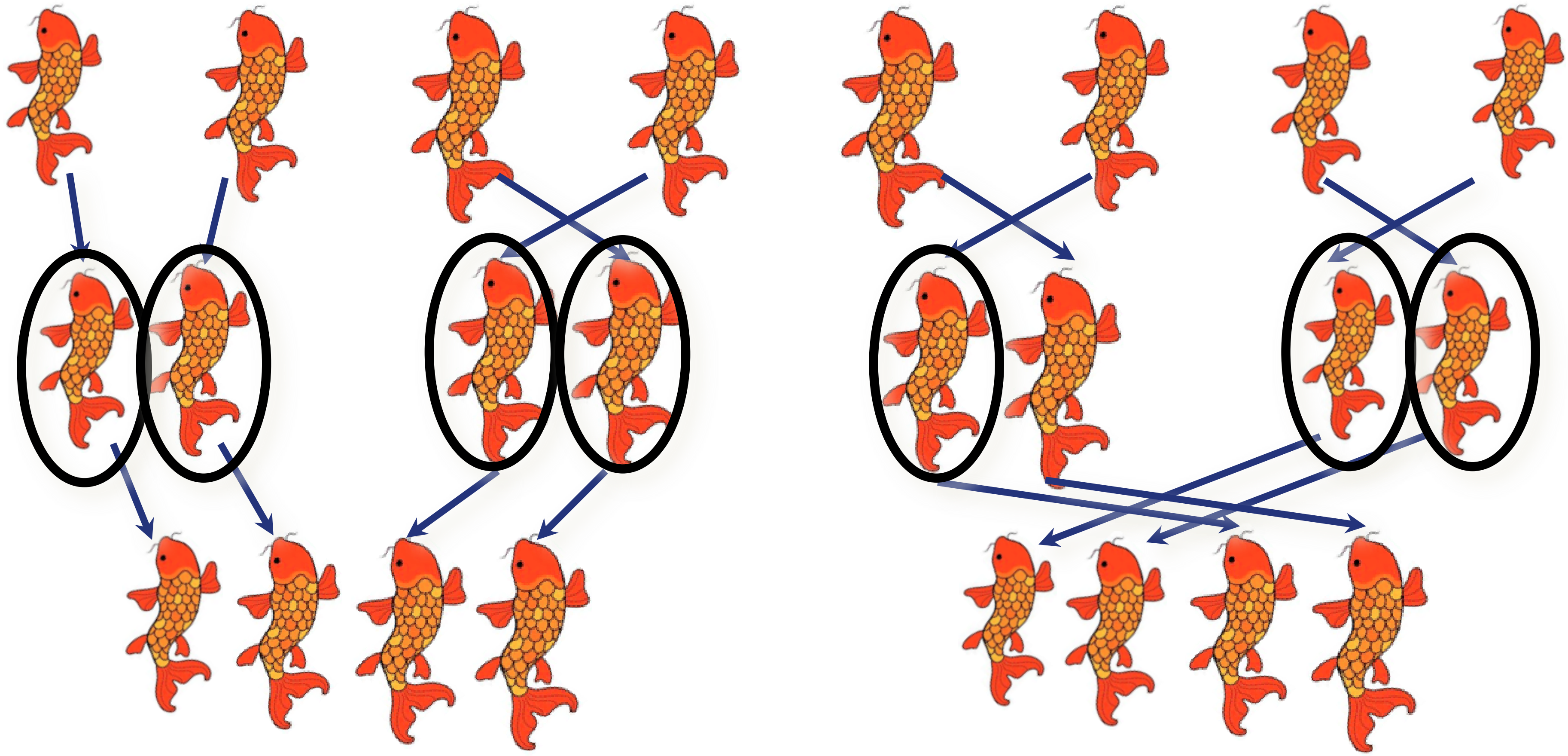


Now each "group" is (trivially) sorted!

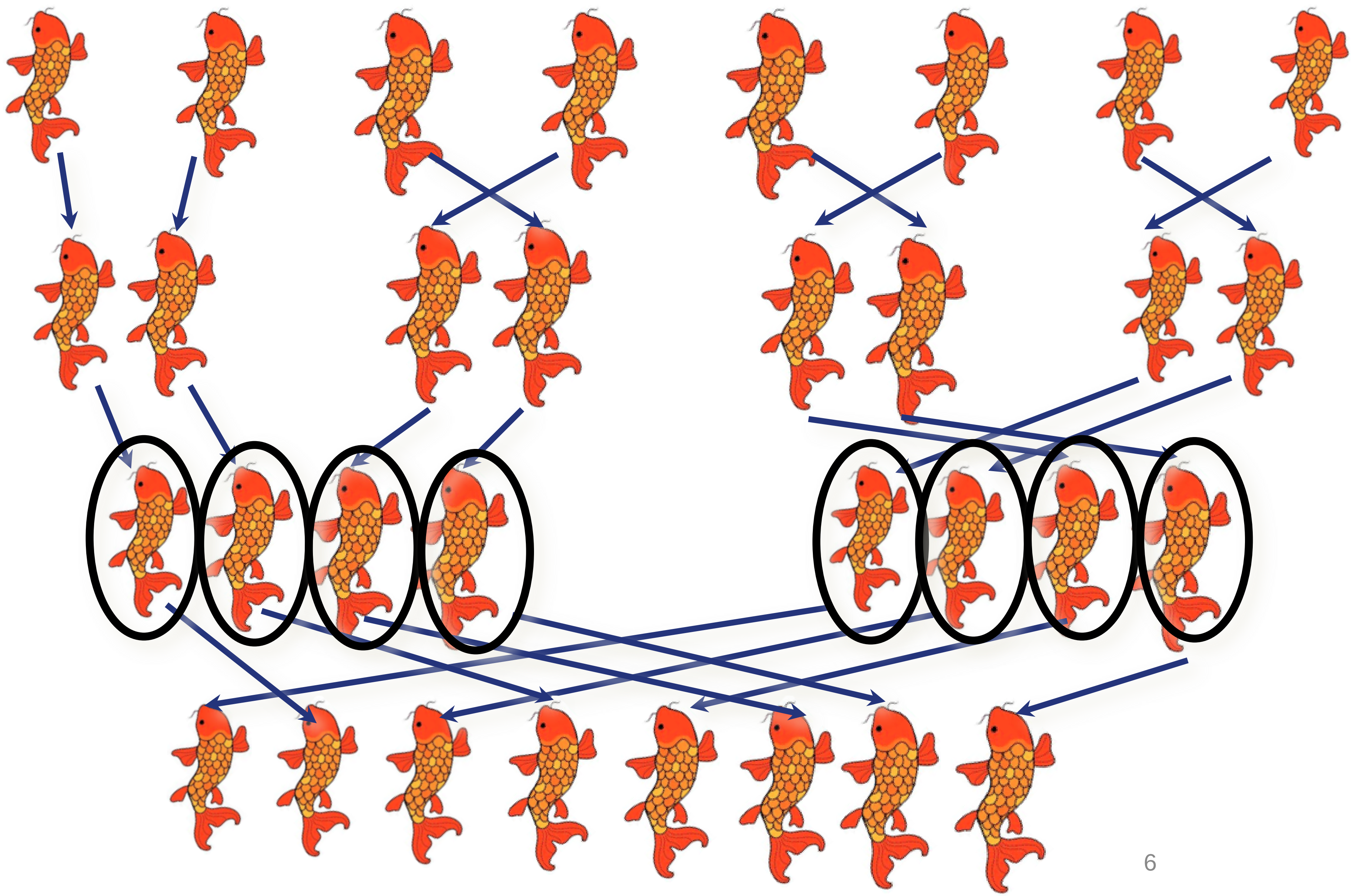
Conquer (merge sorted lists)



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* < length of *a* and *index_b* < length of b
   a. Add the smaller of *a*[*index_a*] and *b*[*index_b*] to the end of *c*
   b. 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*] ≤ *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

```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

list a				list b				list c							
0	1	2	3	0	1	2	3	0	1	2	3	4	5	6	7
12	44	58	62	29	31	74	80	12							
0	1	2	3	0	1	2	3	0	1	2	3	4	5	6	7
12	44	58	62	29	31	74	80	12	29						
0	1	2	3	0	1	2	3	0	1	2	3	4	5	6	7
12	44	58	62	29	31	74	80	12	29	31					
0	1	2	3	0	1	2	3	0	1	2	3	4	5	6	7
12	44	58	62	29	31	74	80	12	29	31	44				

Example 1: Merge (cont’d)

```
list a          list b          list c

 0  1  2  3      0  1  2  3      0  1  2  3  4  5  6  7
12 44 58 62     29 31 74 80     12 29 31 44 58

 0  1  2  3      0  1  2  3      0  1  2  3  4  5  6  7
12 44 58 62     29 31 74 80     12 29 31 44 58 62

 0  1  2  3      0  1  2  3      0  1  2  3  4  5  6  7
12 44 58 62     29 31 74 80     12 29 31 44 58 62 74 80
```

Example 2: Merge

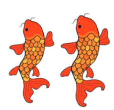
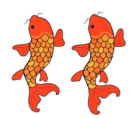
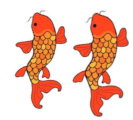
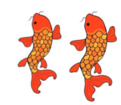
list a	list b	list c
0 1 2 3	0 1 2 3	0 1 2 3 4 5 6 7
58 67 74 90	19 26 31 44	19
0 1 2 3	0 1 2 3	0 1 2 3 4 5 6 7
58 67 74 90	19 26 31 44	19 26
0 1 2 3	0 1 2 3	0 1 2 3 4 5 6 7
58 67 74 90	19 26 31 44	19 26 31
0 1 2 3	0 1 2 3	0 1 2 3 4 5 6 7
58 67 74 90	19 26 31 44	19 26 31 44
0 1 2 3	0 1 2 3	0 1 2 3 4 5 6 7
58 67 74 90	19 26 31 44	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?
what is the maximum number of comparisons?

Example: say we are merging two pairs of 2-element lists:

 with  and  with 

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?

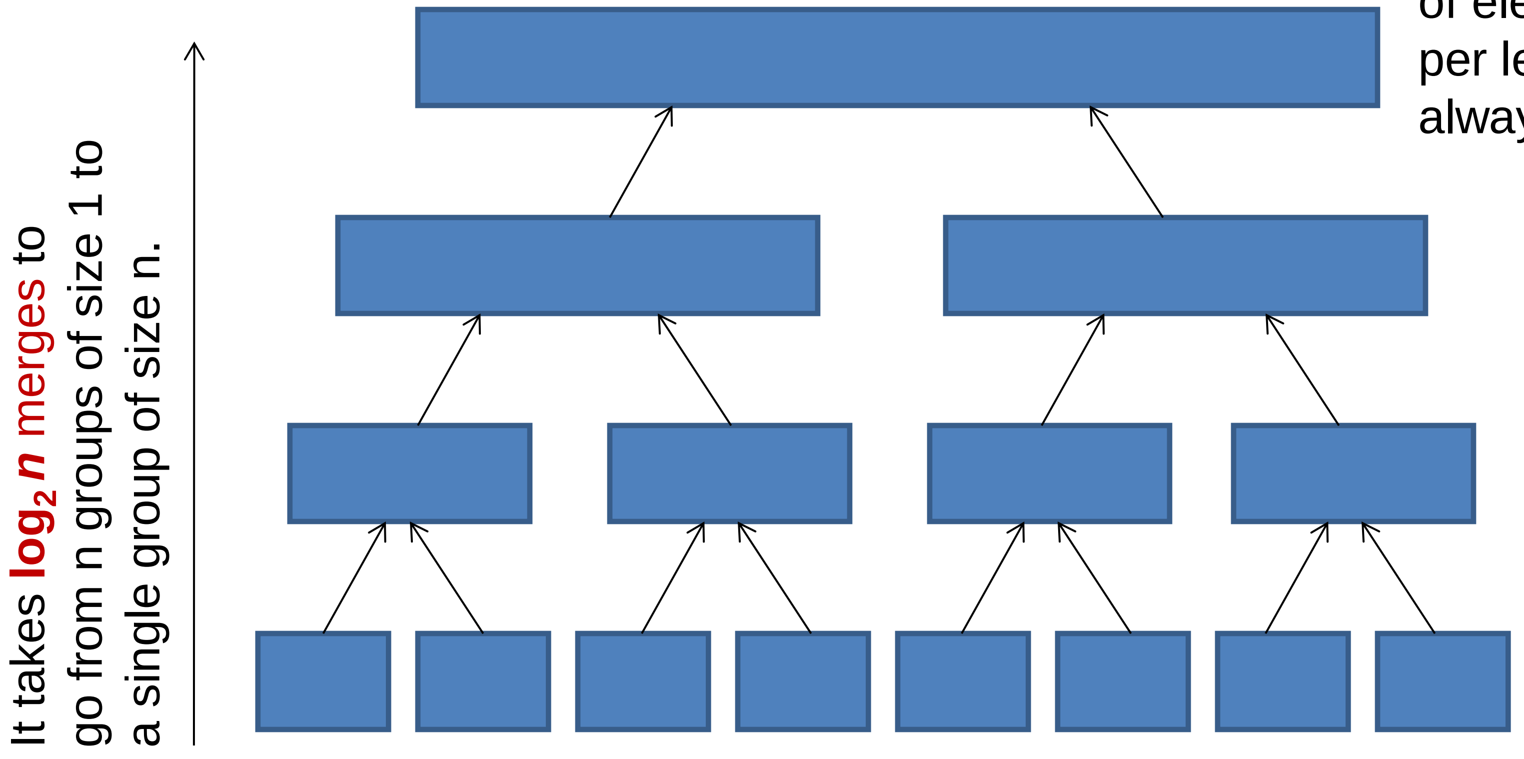
- We saw that each group of merges of *n* elements takes O(*n*) operations.
- How many times do we have to merge *n* elements to go from *n* groups of size 1 to 1 group of size *n*?
- Example: Merge sort on 32 elements.
  - Break down to groups of size 1 (base case).
  - Merge 32 lists of size 1 into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

$5 = \log_2 32$

- **In general: $\log_2 n$ merges of n elements.**

Putting it all together

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



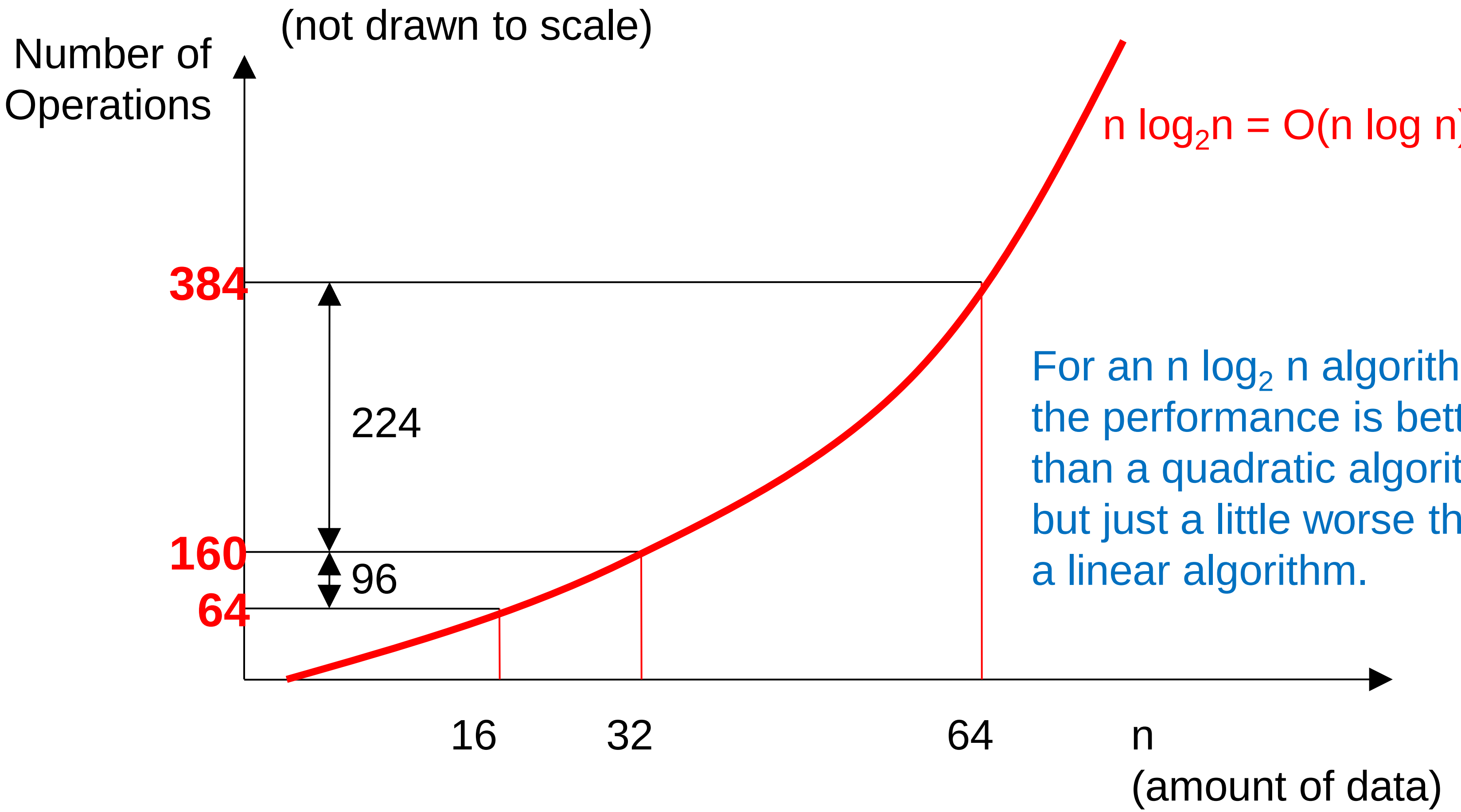
It takes ***n* appends** to merge all pairs to the next higher level.
Multiply the number of levels by the number of appends per level.

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	Order of Complexity
$n \log_2 n$	$O(n \log n)$
$(n + n/2) \log_2 n$	$O(n \log n)$
$4n \log_{10} n$	$O(n \log n)$
$n \log_2 n + 2n$	$O(n \log n)$

O(N log N)



For an n $\log_2$ n algorithm, the performance is better than a quadratic algorithm but just a little worse than a linear algorithm.

Merge vs. Insertion Sort

n	Insertion Sort ($n(n+1)/2$)	Merge Sort ($n \log_2 n$)	Ratio
8	36	24	0.67
16	136	64	0.47
32	528	160	0.30
2^{10}	524,800	10,240	0.02
2^{20}	549,756,338,176	20,971,520	0.00004

Sorting and Searching

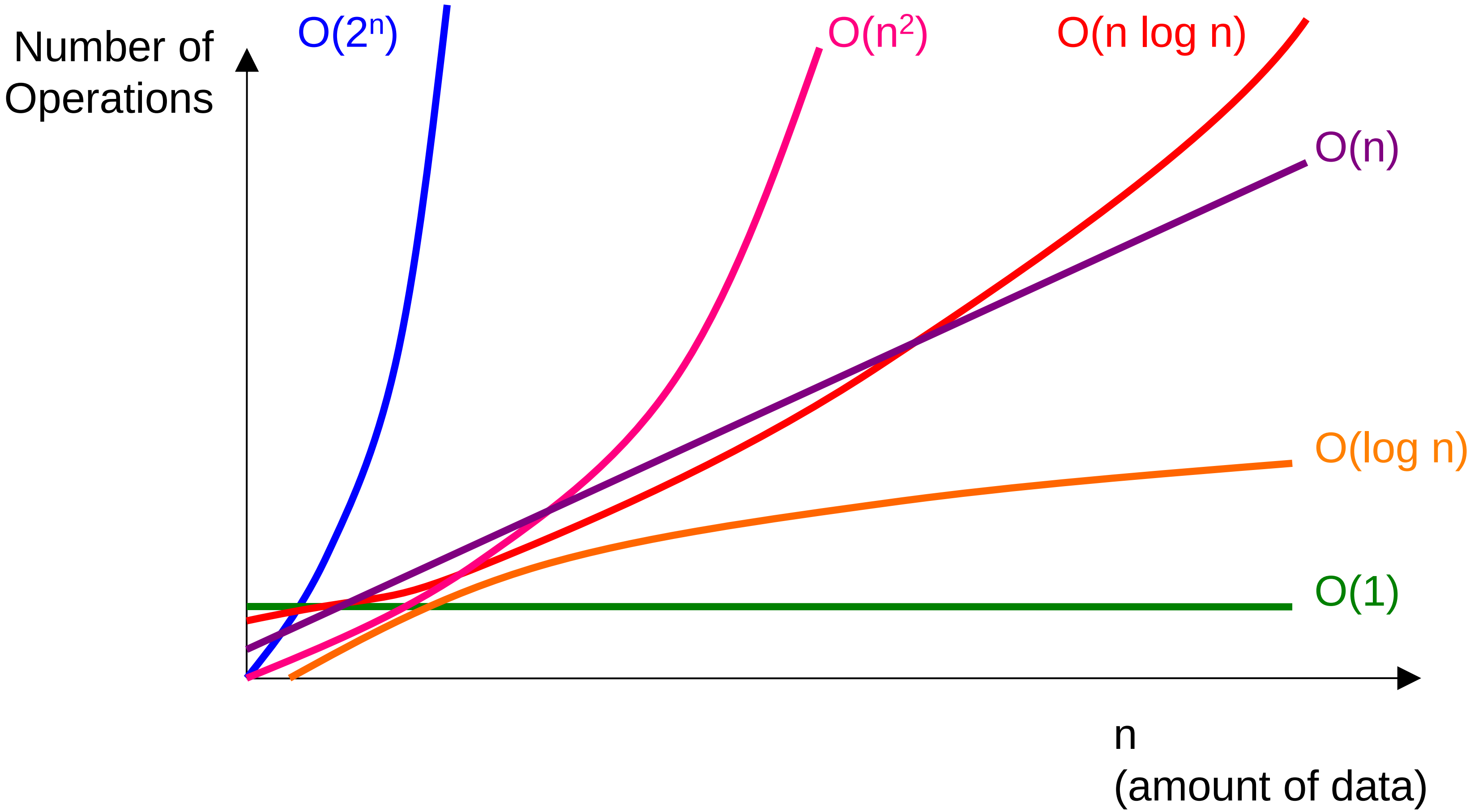
- Recall that if we wanted to use binary search, the list must be sorted.

Insertion sort	$O(n^2)$	(worst case)
Binary search	$O(\log n)$	(worst case)
Total time:	$O(n^2) + O(\log n)$	$= O(n^2)$

- 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): $O(n \log n) + O(\log n) =$ **$O(n \log n)$**

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 <u>modifies it while it sorts</u>
  - Python function **sorted**
    takes a list and <u>returns a new sorted list</u>
  - 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" $O(n \log n)$

Next Time

- **Data Organization**