

# Intro to Algorithms

Professor Kevin Gold

# What is an Algorithm?

- An algorithm is a procedure for producing outputs from inputs.
  - A chocolate chip cookie recipe technically qualifies.
- An algorithm taught in a CS class is typically:
  - Correct/optimal - Produces correct or best possible output.
  - Time-efficient - Takes few steps.
  - Abstract - Doesn't depend on a particular computer architecture or data format.
- For example, you will learn several ways to sort things in this class - independent of programming language or data

# Searching an Unordered List (“Linear Search”)

- Suppose we have a list of  $N$  items
  - Like `(45 90 2 1000 -3 7)` if we're programming in Racket
- We also have a target number, like “1000,” that we are looking for
- We can iterate down a list starting from the beginning to end
- If there are  $N$  items in the list, we can analyze the following...
  - **Best case:** what is the smallest number of items we might need to examine?
  - **Worst case:** what is the maximum number of items we might need to examine?
  - **Average case:** what is the average number of items we might need to examine (let's assume all orderings are equally likely)?

# Best, Worst, Average for Unordered Linear Search

- Searching through a list like (45 90 2 1000 -3 7) for a particular value, like 1000...
- **Best case** is, it's the item in front. Then we just have to look at 1 item!
- **Worst case** is, the item isn't even in the list. Then we need to look at all N items.
- **Average case** if all positions are equally likely (and it's in there) is  $(1 + 2 + \dots + N)/N = N(N+1)/2N = (N+1)/2$

# Notice Two Things About The Average Case for Unordered Linear Search

- (1) We had to make assumptions about the distribution of data. There's no reason to think all positions are equally likely, and we didn't even think about the possibility that the item wasn't there.
- (2) It grew *linearly* (directly proportionally) with the size of the input, just like the worst case. If all we cared about was the *kind* of function (linear vs quadratic vs exponential), **the worst case gave us the right answer with less work & fewer assumptions.**

While the average case may seem more accurate, the worst case requires fewer assumptions and is our default tool of analysis

# Ordered Linear Search

(-3 20 82 104 150 1000 1500)

- Suppose now that the input was already *sorted*. Does that improve the running time of linear search?
  - Recall that we're still going from left to right down the list
  - What is the **best case**?
  - What is the **worst case**?
  - What is the **average case** if we're equally likely to need any position in the list?

# Ordered Linear Search

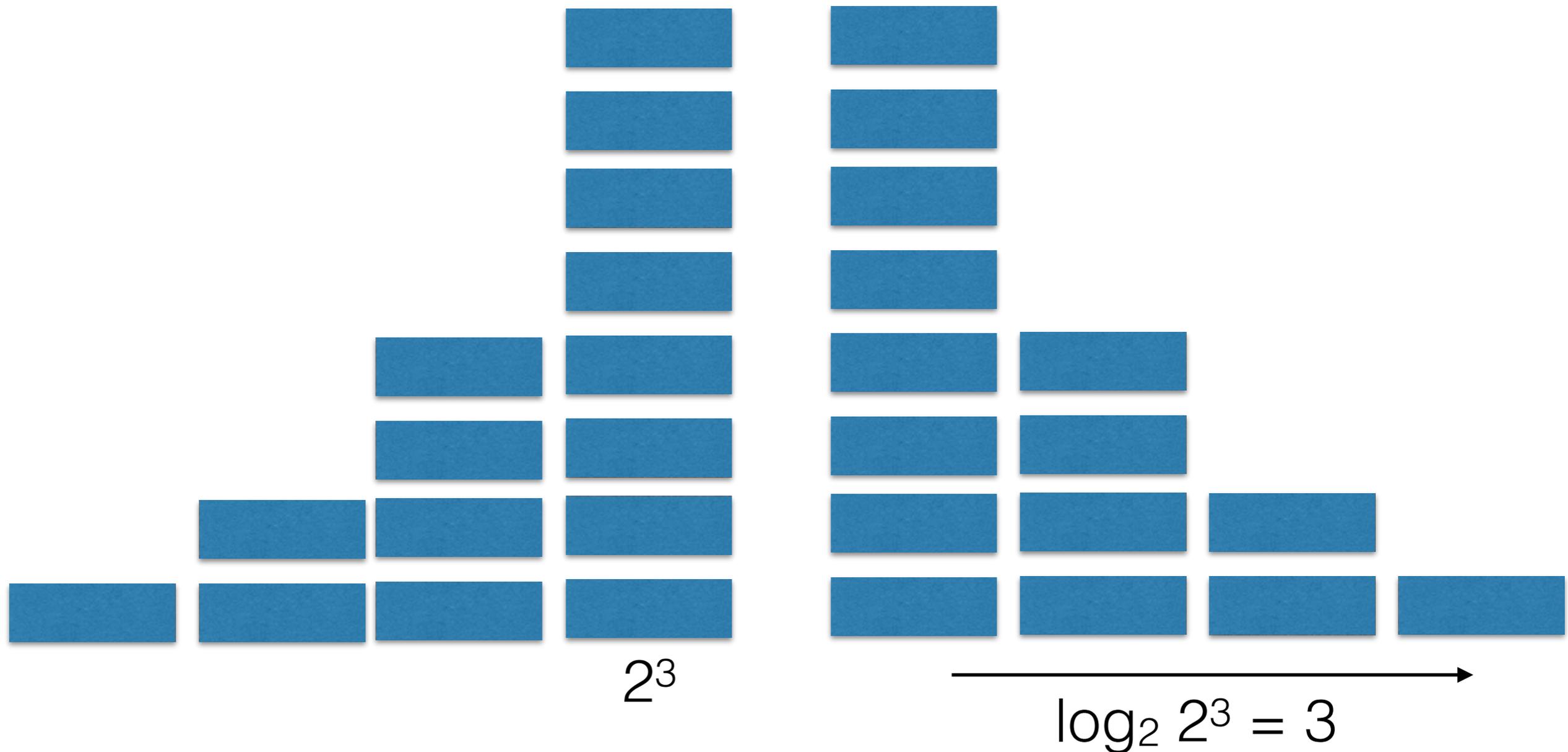
(-3 20 82 104 150 1000 1500)

- The **best case** is still that we happen to need the item at the beginning of the list. 1 operation.
- The **worst case** is still that the item is not in the list at all (or that it's last).  $N$  operations.
- The **average case** is again the average over all the positions,  $(N+1)/2$

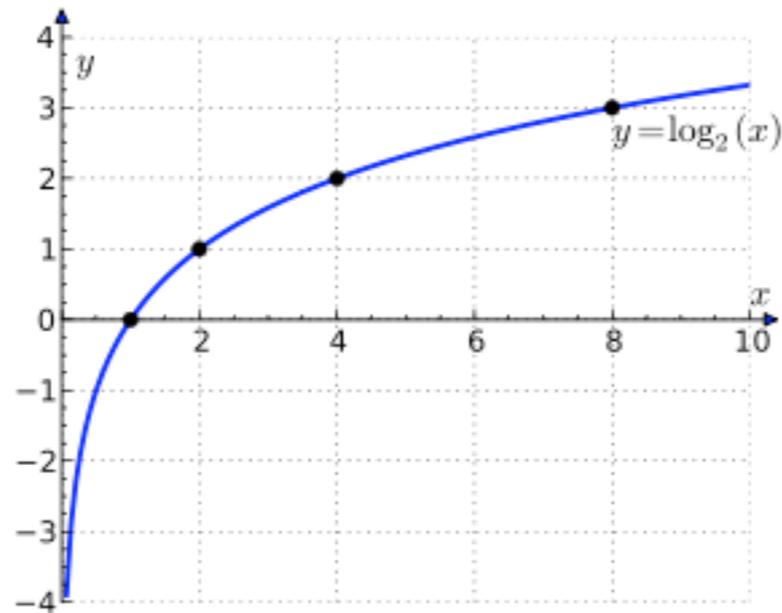
# Binary Search

- Binary search is genuinely faster than linear search
- It must be performed on a **sorted array** (like a list but slightly different)
- The idea is to “throw out” half the values with each iteration, instead of “throwing out” one.
- If there are  $N$  values to start, after one round  $N/2$  remain, then  $N/4$ , then  $N/8$  ...
- The total rounds in the worst case are roughly  $\log_2 N$  - the power of 2 you would need to hit  $N$  (e.g.  $\log_2 256 = 8$ ). This is *much* better than linear

# Recall: Logarithms are Exponentiation in Reverse



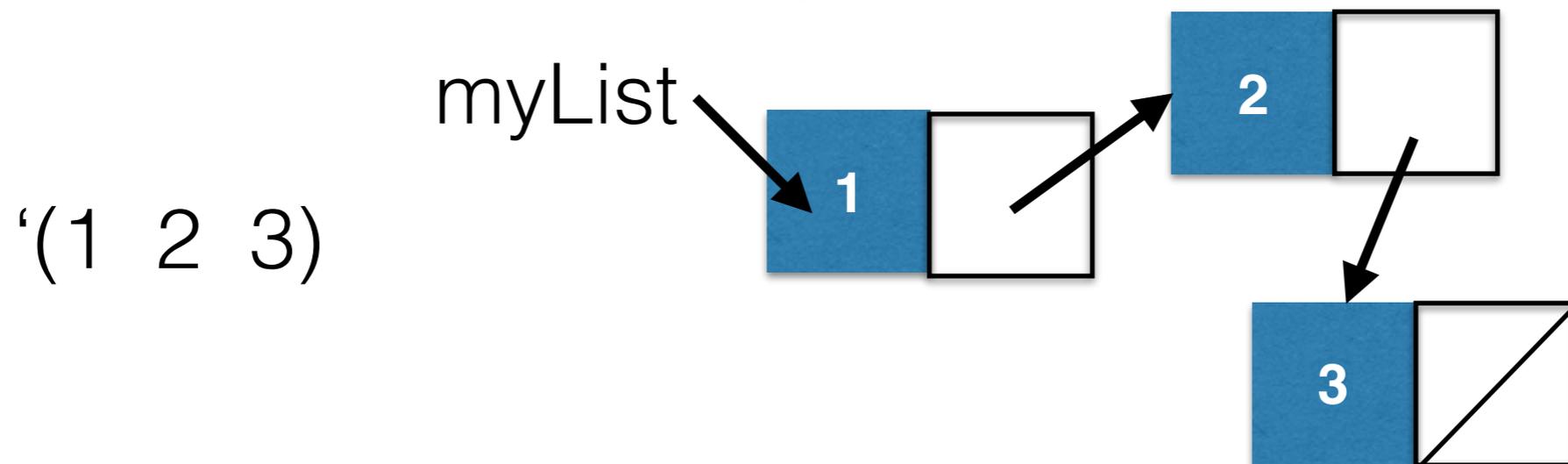
# Logarithmic Growth is Great for Algorithm Running Time



- Taking the log of gigantic numbers like  $\log_2 1,000,000$ , we get reasonable numbers like  $\sim 20$
- In fact, compared to *any* linear function ( $y = an + b$ ), a logarithmic function will “cost less in the long run”

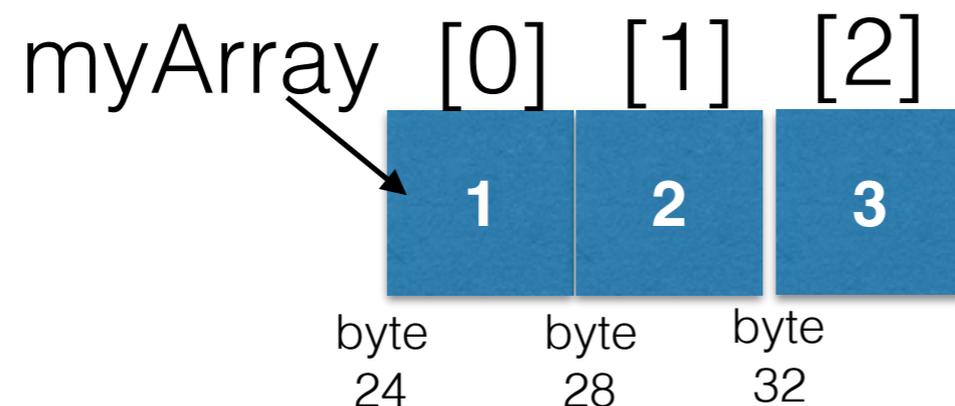
# Arrays versus Linked Lists

- In analyzing algorithms, it matters how the underlying data is stored. Some operations are fast for one data structure, slow for another.
- *Linked lists* are composed of pieces of data that each “point” to the next piece in memory, which could be far away. The middle of a list is accessed by following these pointers around - meaning, no fast access for the middle.



# Arrays versus Linked Lists

- In analyzing algorithms, it matters how the underlying data is stored. Some operations are fast for one data structure, slow for another.
- *Arrays* are blocks of memory where pieces of data sit side-by-side. The program can compute quickly exactly where each piece of data is, because it's not scattered across memory. Accessing the middle doesn't require following links - it is "constant time."

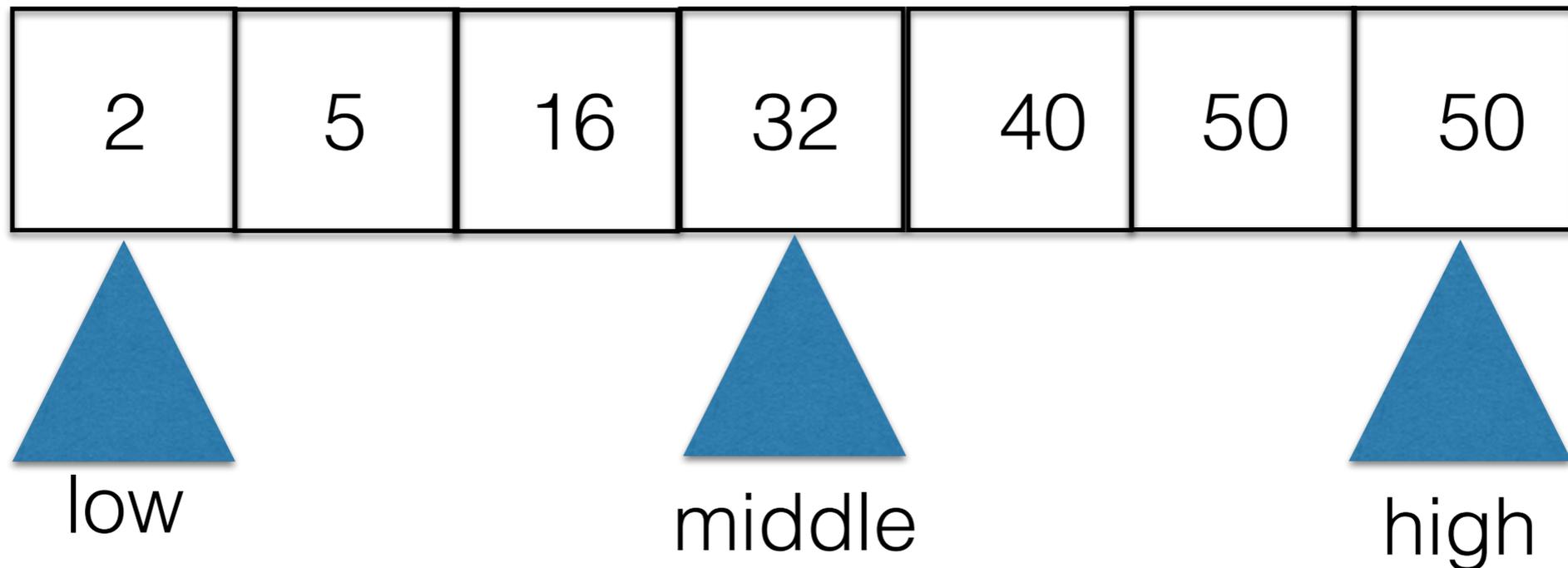


myArray[2]  
*myArray address "24"  
each integer is 4 bytes  
 $24 + 4 * 2 = 32$   
for [2] look at byte 32*

# “Constant Time”

- Any number that does not scale with the size of the input (or other parameters of the problem) is a **constant**. 2, 0.5, 10 ...
- If an operation takes “constant time,” that means the operation takes the same amount of time no matter how much input there is.
  - Array access is constant time. No matter how much input there is, it's multiply, add, access.
  - An algorithm can't really get better than constant time.
- One reason we don't get more specific than “constant” is that without a specific architecture, we don't really know how long a particular operation will take anyway

# Binary Search, Pictorial Version



We keep track of the “low” and “high” ends of the search.

Suppose we’re looking for 5.

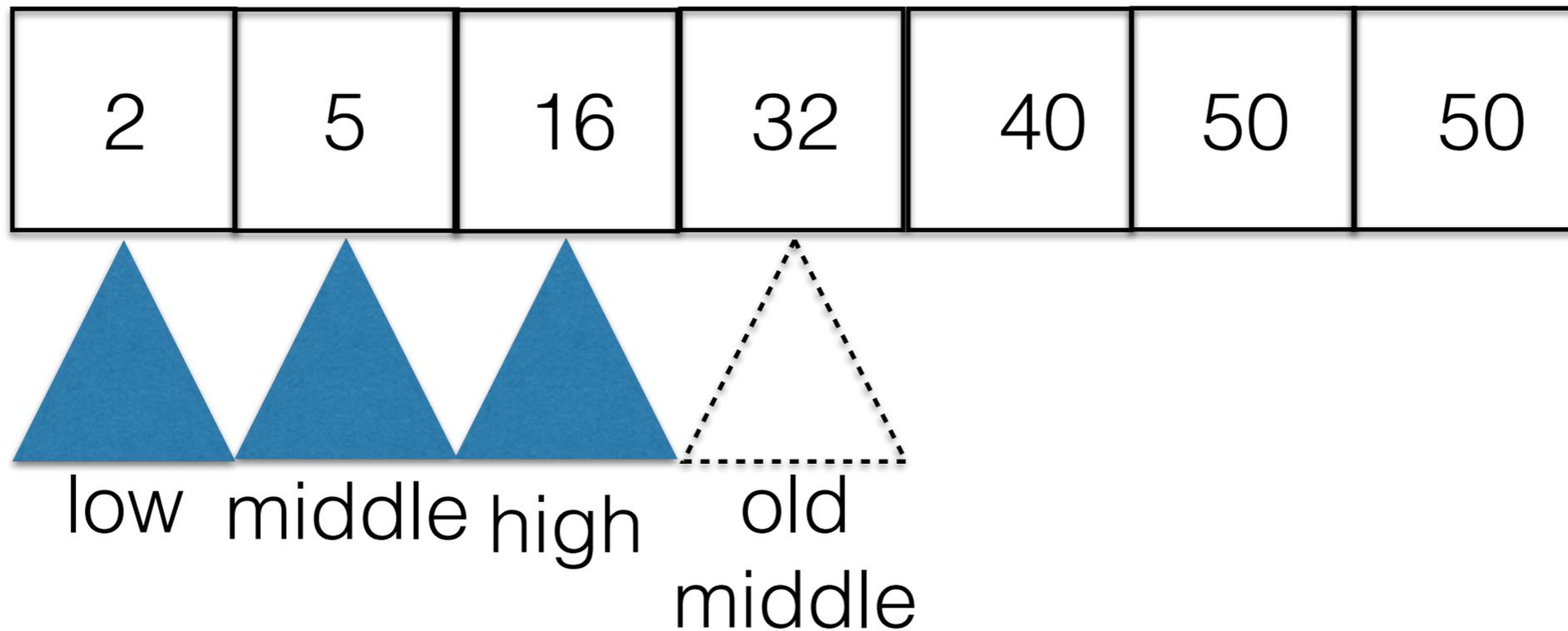
We first ask, “Is the middle 5?”

It is not.

Is 5 less than or greater than the middle, 32?

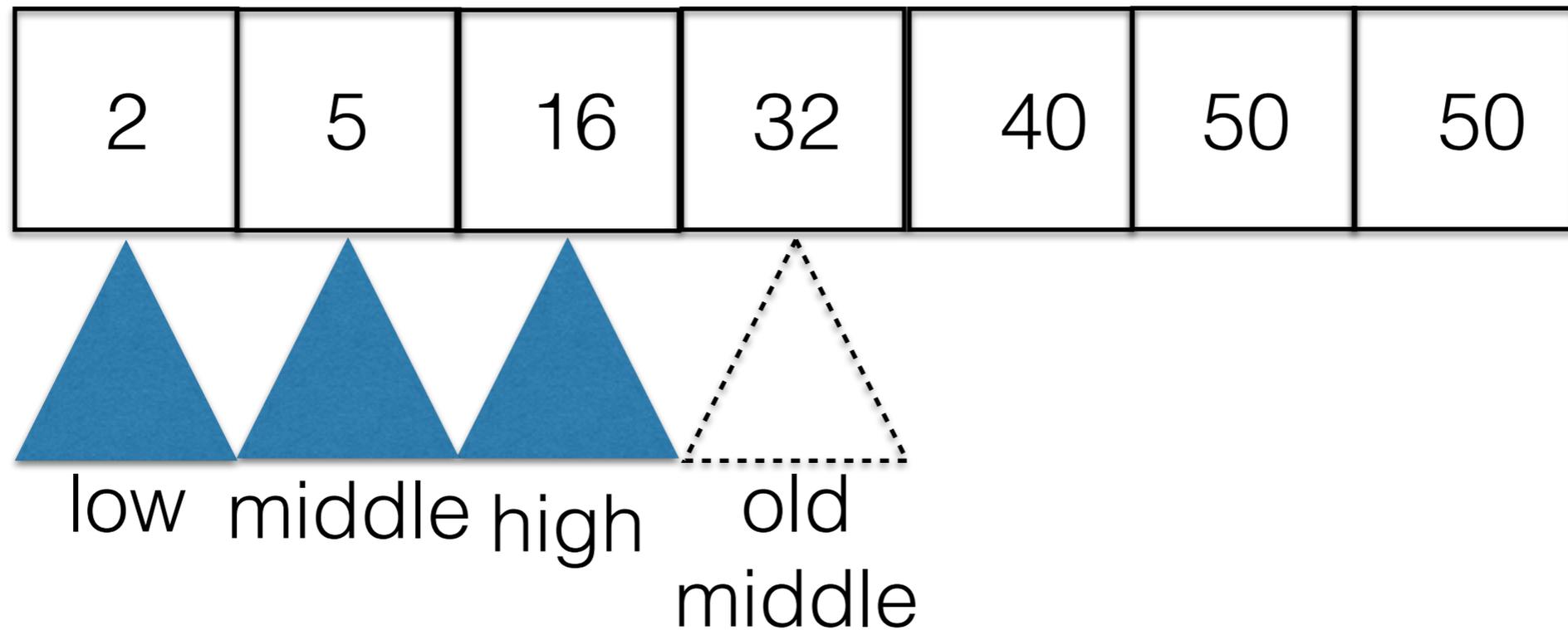
It is less.

# Binary Search, Pictorial Version



The new highest possible value is left of the old middle.  
Is the middle what we're looking for now?  
In this case, it is - we're done.

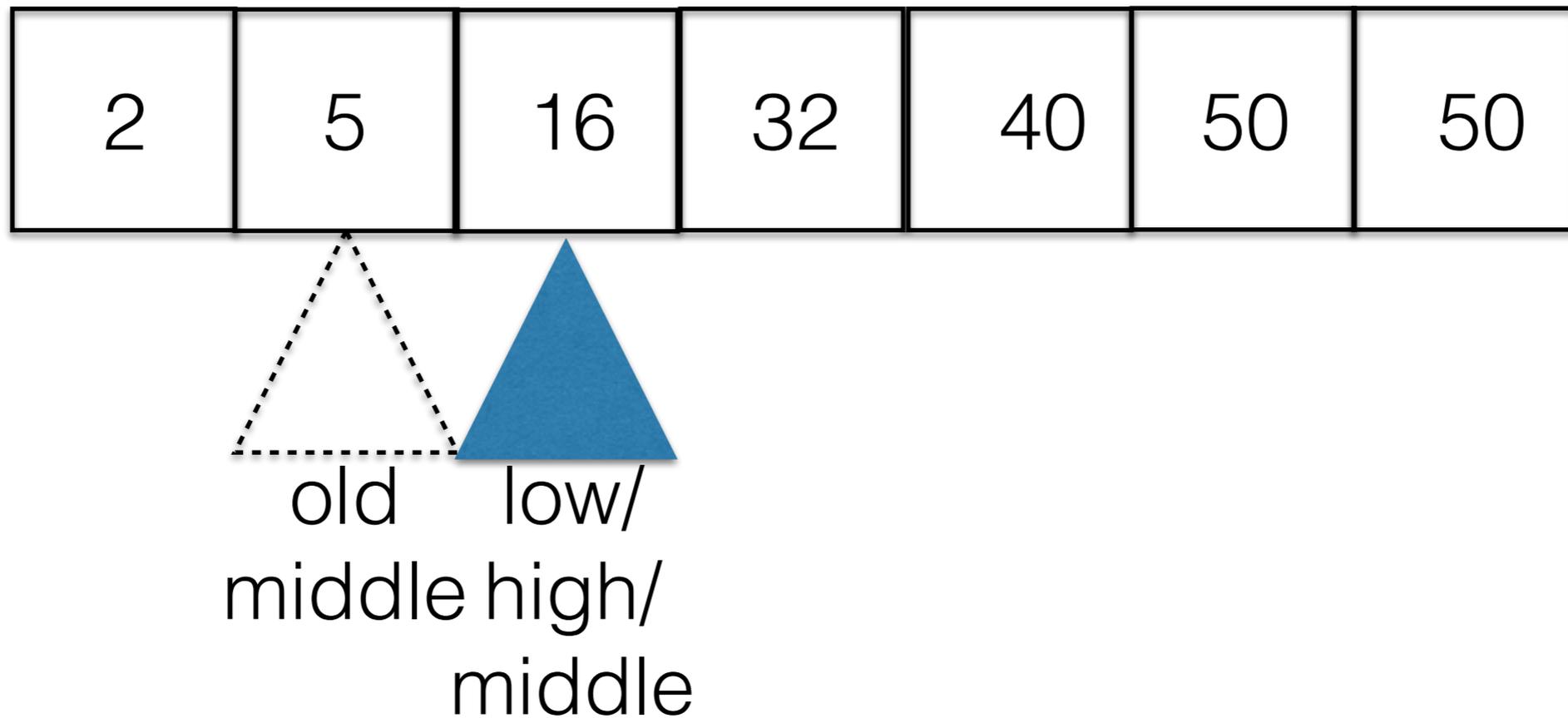
# Binary Search, Pictorial Version



Suppose we had been looking for 7 instead.

The first step is the same,  
but now, we want to look right of the 5.

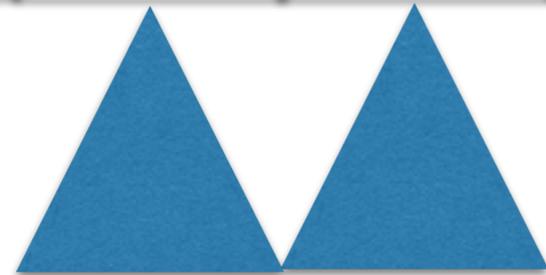
# Binary Search, Pictorial Version



Now  $low == high$ , so there's just one more place to look.  
But 16 isn't 7.

# Binary Search, Pictorial Version

2	5	16	32	40	50	50
---	---	----	----	----	----	----



high    low/  
         old  
         middle

Since  $7 < 16$ , we move “high” left of “middle.”

But now “high” is left of “low.”

This is how we know the value isn't here.

# “Pseudocode”

- When thinking about algorithms, we want to focus on big ideas, not get bogged down in syntax
- “Pseudocode” refers to sketches of programs, that aren’t in any real language
- There are common conventions for pseudocode, but we’ll defer to your algorithms class for those
- Be aware that pseudocode is usually written in an “imperative” style similar to Java, Python, or C, where commands are written one after the other (as opposed to Racket’s “functional” style that likes to nest and recur).
- `//` indicates a comment until end of line (C/Java style)

# Binary Search (Recursive)

BINARY-SEARCH(A, key, low, middle, high):

if low > high:

    return NOT\_FOUND

else if A[middle] == key:

    return middle

else if A[middle] > key:

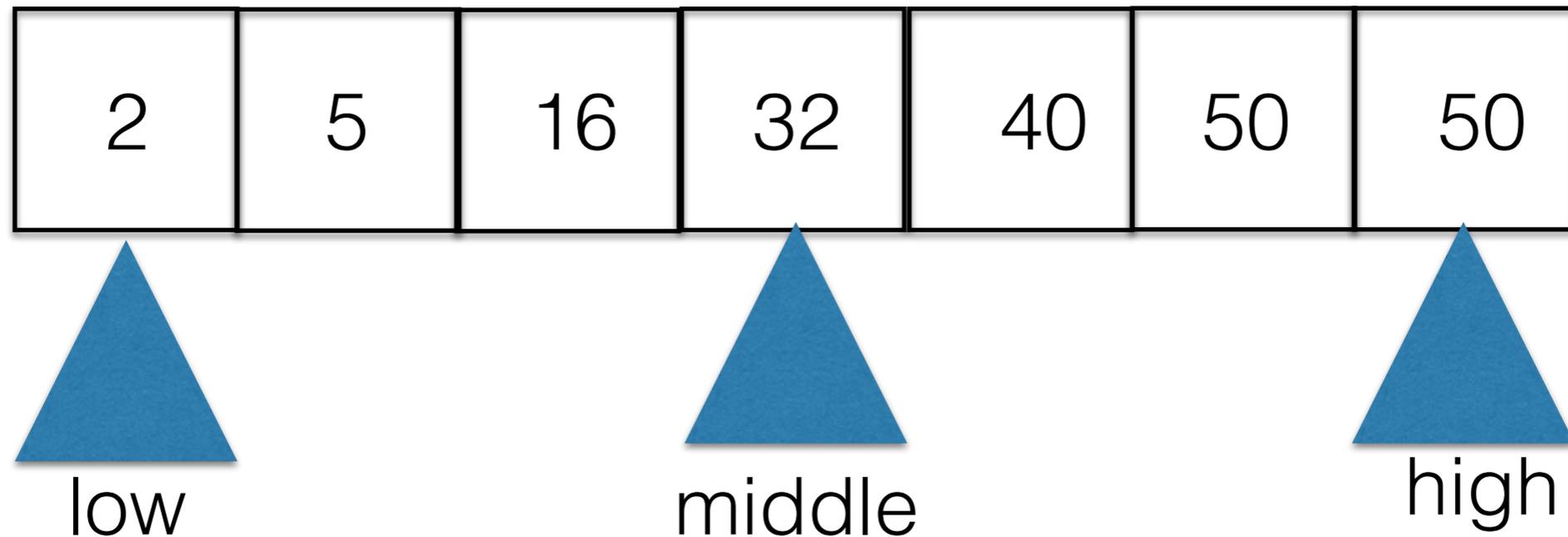
    return BINARY-SEARCH(A, key, low, floor((low+middle)/2), middle-1) // recur on 1st half

else: // A[middle] < key

    return BINARY-SEARCH(A, key, middle+1, floor((middle+high)/2), high) // recur on 2nd half

A is an array,  
key is what we're looking for,  
"low" init to first place in array,  
"high" init to last place in array,  
"middle" init to (low+high)/2

# Try some binary search



- How many rounds to discover “40” is in the array?

# Linear Search vs Binary Search

- We tried fiddling with linear search a little to try to make it more efficient (by sorting it), but it took a very different approach to make big speed gains.
- This is typically how algorithms work - a clever different approach will produce better results than minor optimizations to the existing algorithm

# Sorting

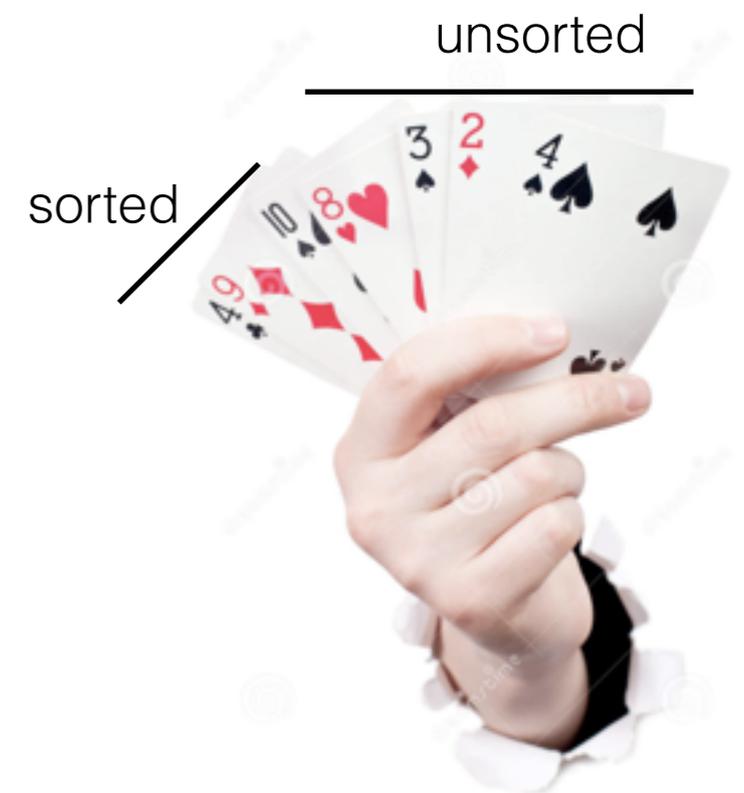
- There are **many** ways to sort things, but only a few are popular
- Two of the methods that we will see, **insertion sort** and **selection sort**, are similar to what you might try to implement yourself if you had to invent something
  - But we will try to convince you they're not great
- A third way we'll cover, **mergesort**, is more efficient than the other two, and is elegant in a way.
  - But it can also be tricky to wrap your head around it
- Other famous sorts include **bubblesort** (concise to code but not great) and **quicksort** (popular and efficient, similar to mergesort but more complex)

# Some Efficiency Notes for Sorts

- Often, **comparisons** of elements are the most time-consuming operations in practice for a sort, so we just count those.
  - For example, string comparisons may require looking at all the characters.
- **Assigning** values to particular spots, **accessing** particular values, and **swaps** of two elements are very **efficient for arrays**, but not for linked lists. Array-based sorts usually use swaps.
- **Insertion** of an element between other elements is **fast for linked lists**, but slow for arrays (all items past that point must be scooted down to make room). Linked list-based sorts insert efficiently.

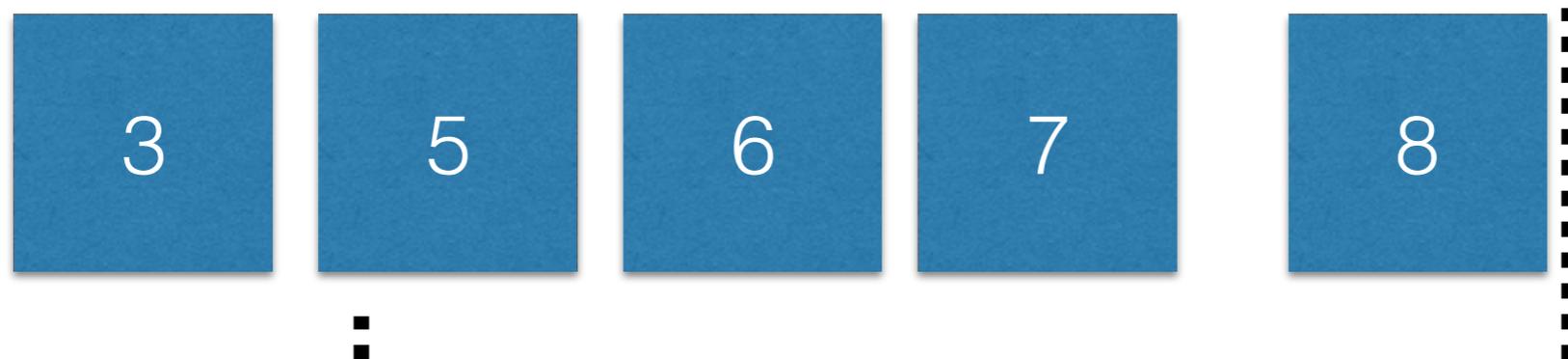
# Insertion Sort Notes

- Insertion sort works similarly to how you might sort a hand of cards
  - Maintain a “sorted” part and “unsorted” part
  - One-by-one, stick an unsorted card into the right place in the “sorted” part
- But since we’re dealing with arrays, there’s no quick visual scan of where an item ought to go - so we shift the item left until its neighbor to the left isn’t bigger



# Insertion Sort

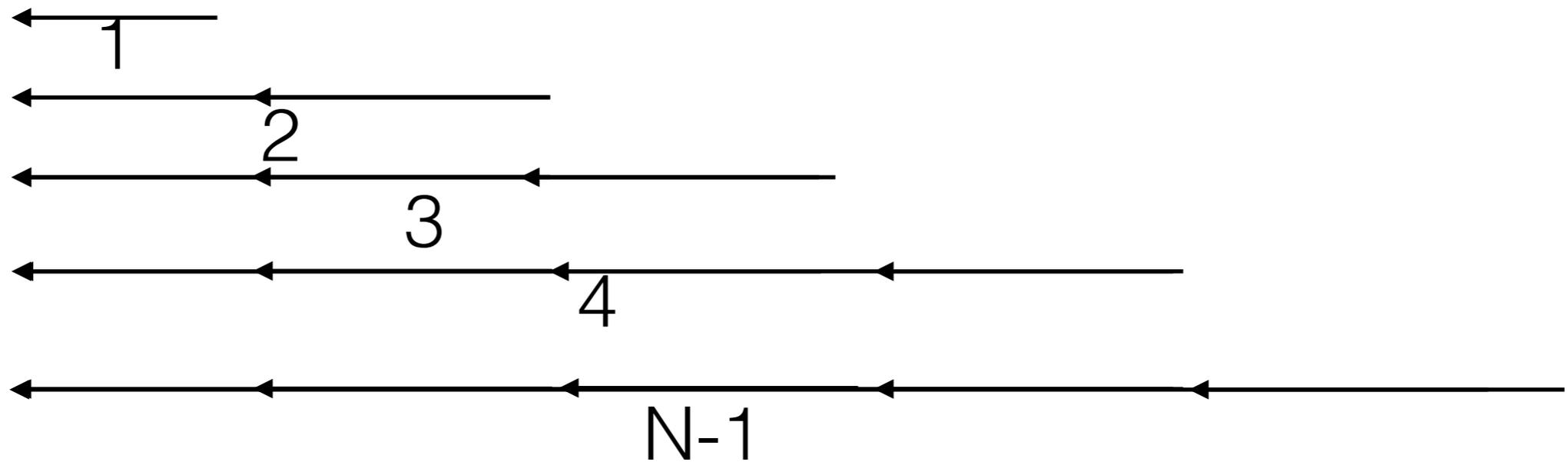
FROM LEFT TO RIGHT, SWAP EACH ELEMENT LEFT  
INTO SORTED ORDER  
(I.E. STOP WHEN LEFT NEIGHBOR IS  $<$  OR  $==$ )



= line of sortedness; stuff to its left is sorted

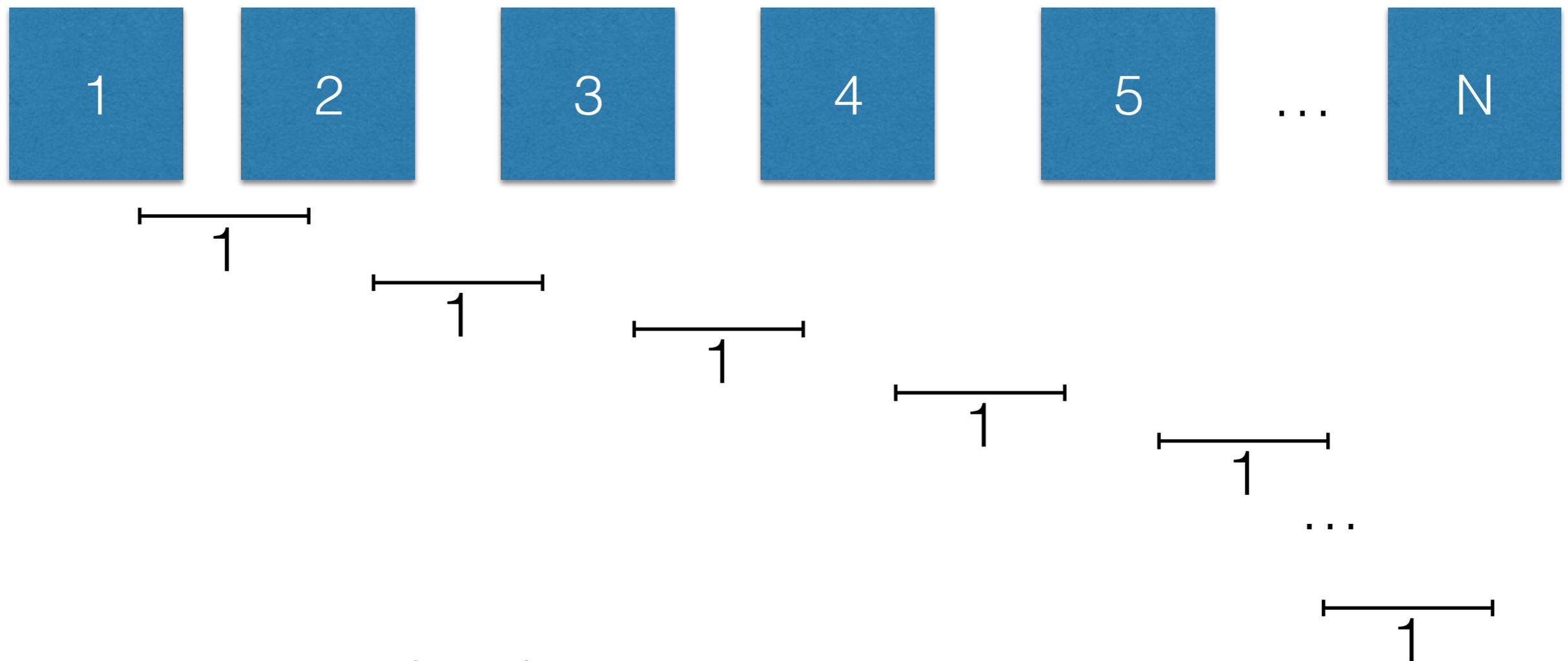
*We are INSERTING each item between the already sorted items*

# Insertion Sort Worst Case: Reverse Order



$$1+2+\dots+N-1 = (N-1)N/2 \approx N^2$$

# Insertion Sort Best Case: Already Sorted



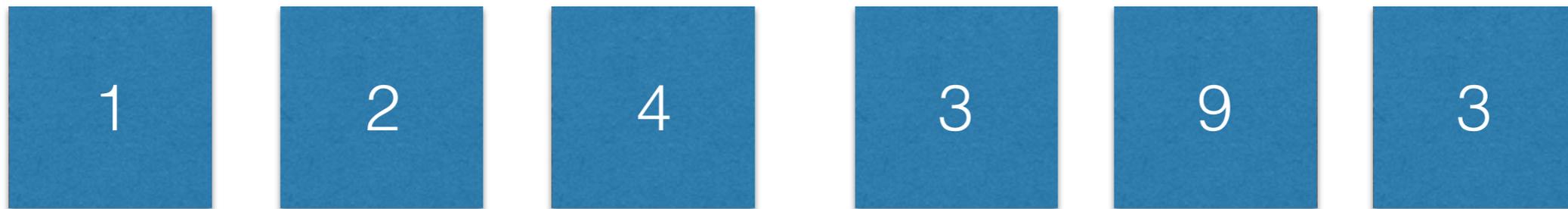
$$1 * (N-1) = N-1 \approx N$$

# Is Insertion Sort Better With Linked Lists?

- We might wonder whether insertion sort is more efficient with linked lists (no need to swap things down)
- For each item in original list,  
    Insert in sorted order in target list T
- Inserting in the correct place in T still requires iterating down the list - in the worst case, we iterate down to the end of the growing list every time
  - First insert is free, then one comparison, two, three ...
- $(1 + 2 + \dots + (N-1))$  comparisons is still about  $N^2$  operations

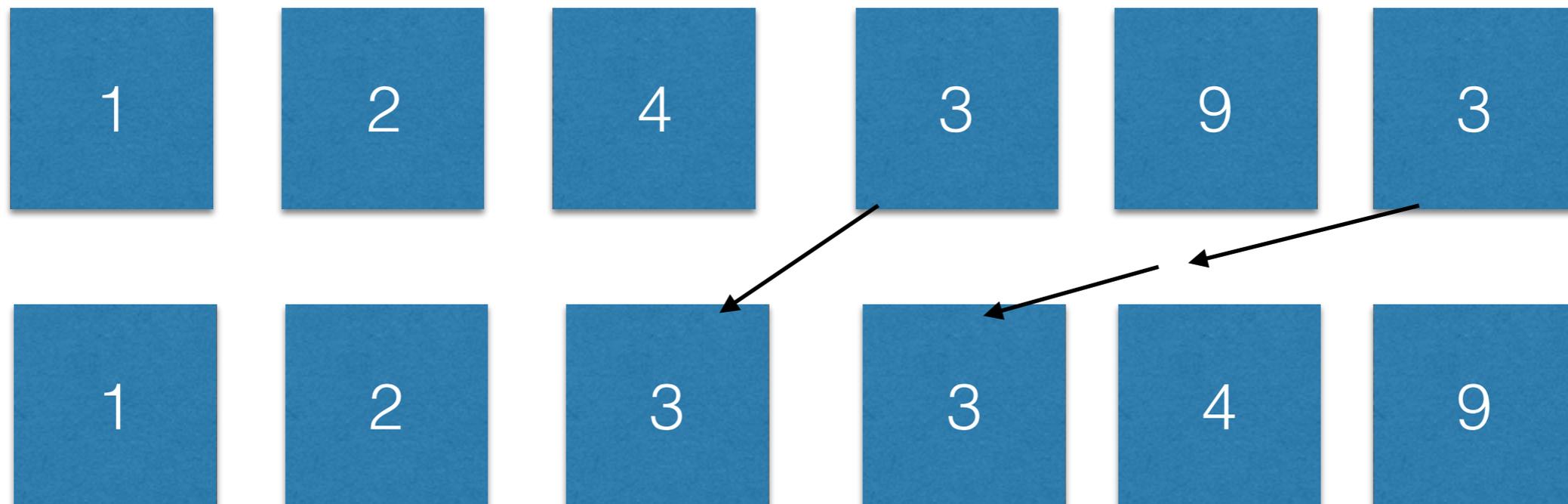
# Try An Insertion Sort

- Run on the array 1 2 4 3 9 3. How many swaps? Could we figure out the number of comparisons from that?



# Try An Insertion Sort

- Run on the array 1 2 4 3 9 3. How many swaps?  
Could we figure out the number of comparisons from that?
- 3 swaps, and each of the  $n-1$  items to the right also has one comparison where nothing happens. 8 comparisons.



# Selection Sort

## Overview

- You may have scanned a list or array for a minimum before - for each item, compare it to bestMinSoFar
- Selection sort extends this idea to a whole sort. Find the minimum, then find the minimum of what's left, then find the 3rd smallest ...
- We'll see an array-based selection sort, where swaps are efficient

# Selection Sort

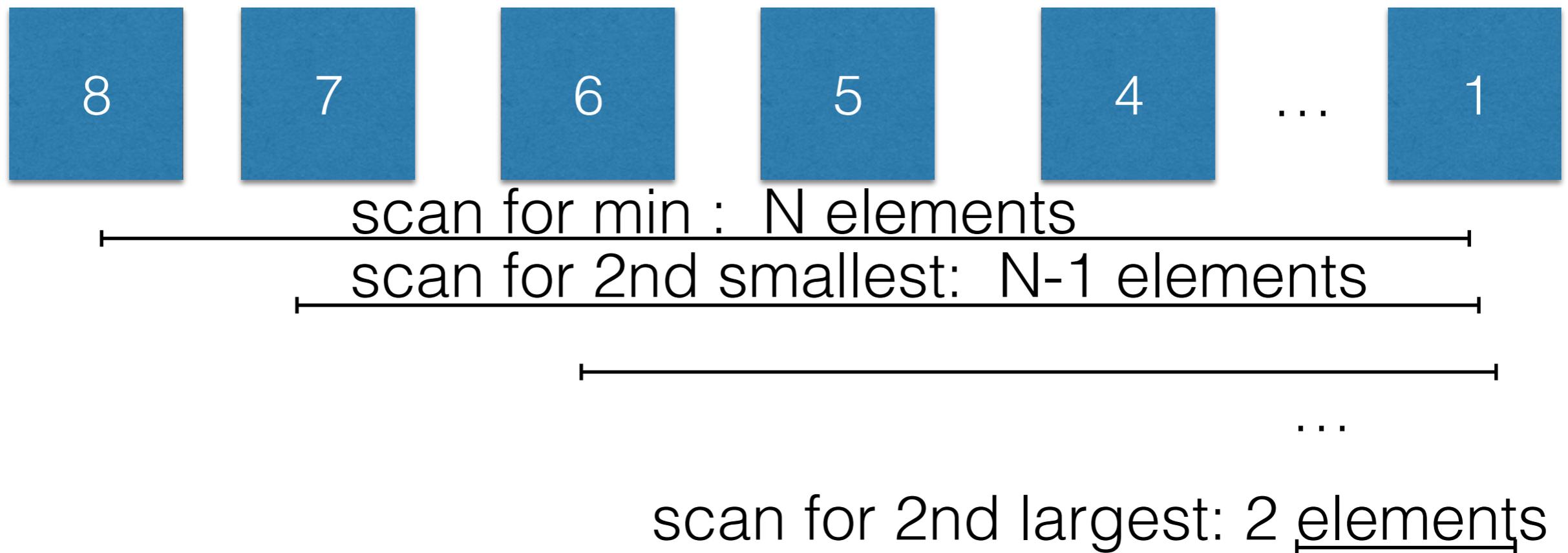
SEARCH FOR MIN, SWAP INTO PLACE,  
REPEAT WITH 2ND SMALLEST, 3RD SMALLEST...



 = line of sortedness; stuff to its left is sorted

*We are SELECTING the item we want next in the order*

# Selection Sort Analysis (Worst, Best, & Average)



$$N+(N-1)+(N-2)+\dots+2 = (N-1)(N+2)/2 \approx N^2$$

# Mergesort Overview

- Mergesort is our first recursive sort - it calls mergesort itself to sort smaller parts of the list, then merges the results into one big sorted list.
- A “merge” takes two sorted lists and combines them into a single sorted list.
- The pseudocode is therefore pretty concise:

Mergesort(L):

If  $\text{length}(L) \leq 1$ :

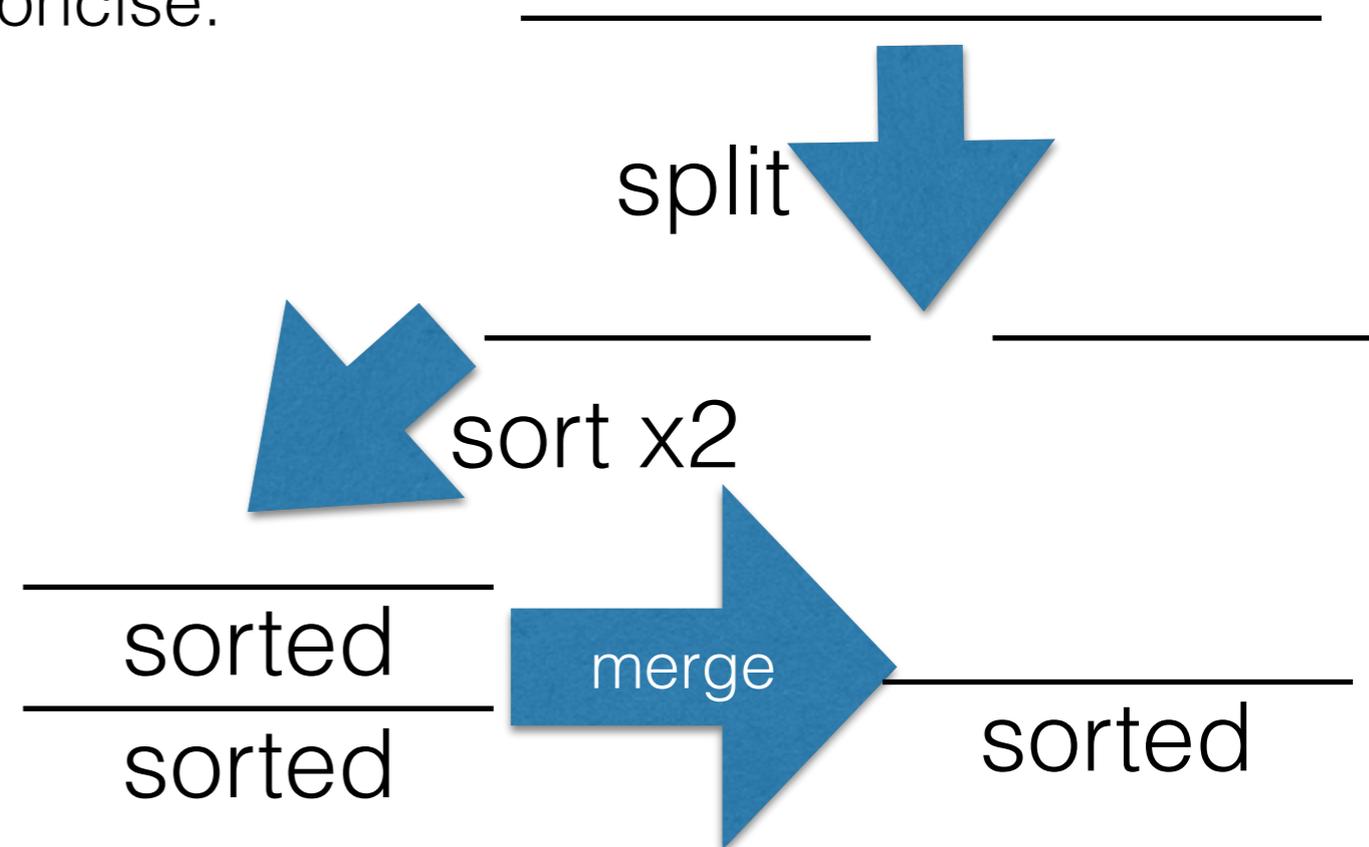
    return L // nothing to sort

Else:

    A = Mergesort(first half of L)

    B = Mergesort(second half of L)

    return merge(A,B)



# Merge

Input: 2 **sorted** lists

Output: One sorted list containing all elements

Running time: Linear in number of all elements

While elements in both lists remain,

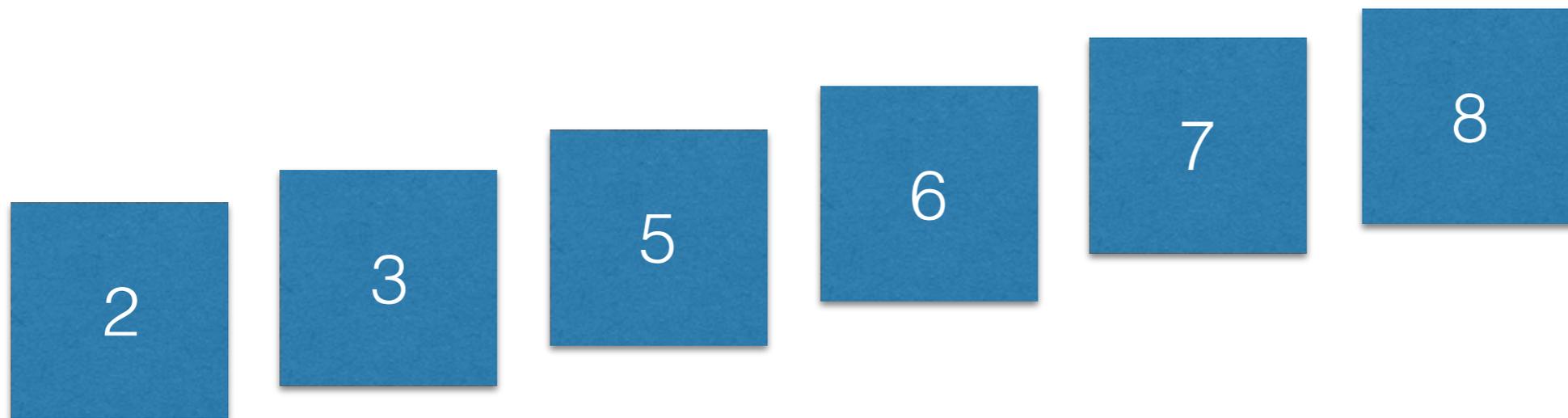
Compare the smallest elements of both lists.

Remove the smaller and append it to the output.

If elements of one list remain,

Append them to the end.

Return the output.

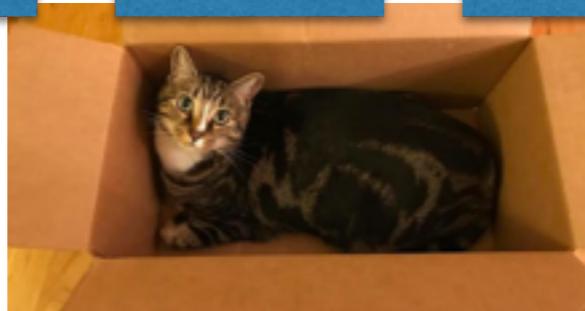


# Mergesort

If the list is has fewer than two elements, return it. Otherwise:  
Mergesort the first half of the list.  
Mergesort the second half of the list.  
Merge the sorted halves.



Recursion  
Level 1



Recursion  
Level 2

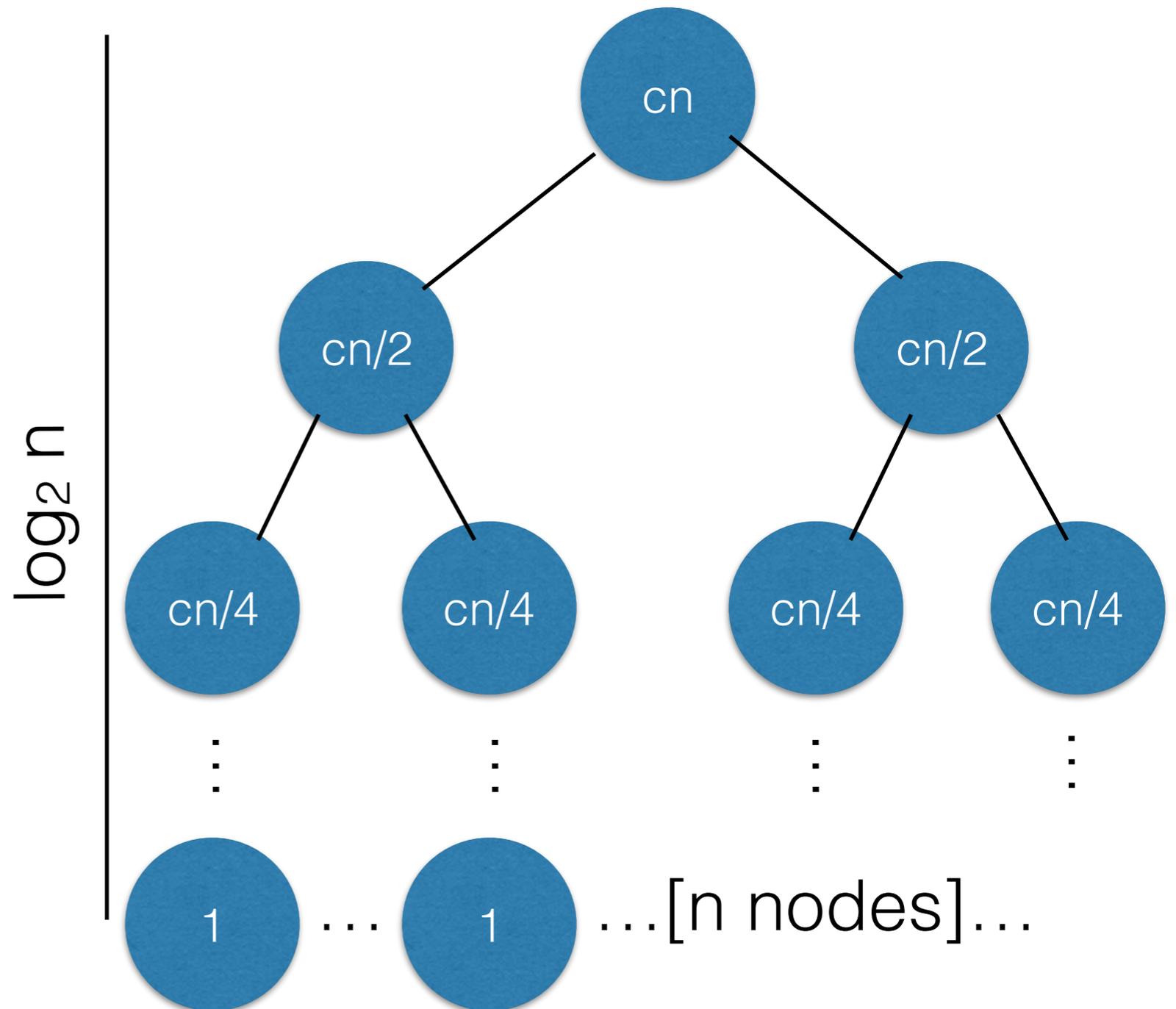


Recursion  
Level 3



# Mergesort Analysis With Recursion Tree

- Since the work done is in the form of “work at this level plus work at lower levels” we can visualize the work as a tree
- With a bound of  $cN$  work per level and  $\log_2 N$  levels, the total work is  $cN \log N$ , where  $c$  is some constant.

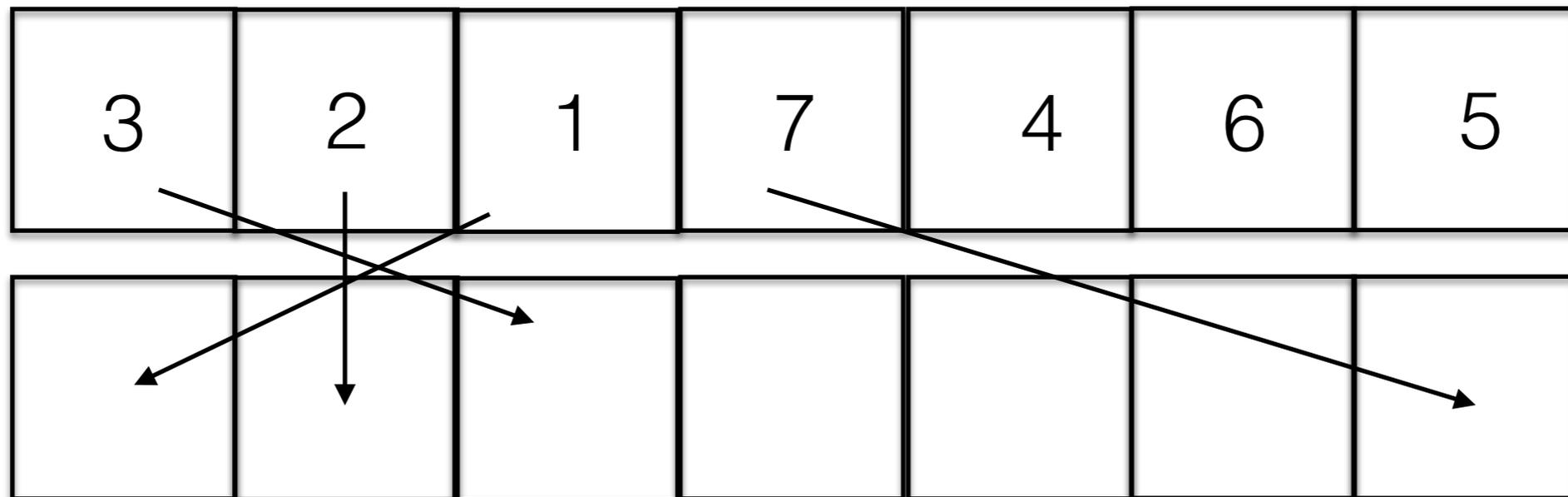


# $N \log N$ is as Good as We Get for Comparison-Based Sorts

- All three of these algorithms are **comparison-based**, meaning they determine the final ordering through pairwise comparisons of elements
- It's possible to prove that you **can't do better than mergesort's  $N \log N$  time using a comparison-based sort**
  - Proof sketch: There are  $N!$  possible correct orders, so fewer comparisons wouldn't be able to tell the difference between some of them
- However, sorts that can tell where data should go just by looking at it (not comparing) can achieve better times

# Example of a Linear Time Sort

- Suppose I have an array of  $N$  unordered values - the integers 1 through  $N$ , inclusive
- In a single pass, through the  $N$  values, I can assign them to the correct places in the target array
- That's time on the order of  $N$ , not  $N \log N$  - it was because I was able to put items directly where they need to go
- **Bucket Sort**, **Counting Sort**, and **Radix Sort** are all sorts that are not comparison-based, but put things where they go in linear time - but the data must be "special"



# How We Approximate Running Times

- It is typical to approximate algorithm running times to just use  $N^d$ , where  $d$  is the highest exponent of  $N$  in the running time.
  - Thus Insertion Sort and Selection Sort are both “about  $N^2$ ”
  - Linear search is “about  $N$ ”
- We pay attention to additional log factors, but don't care about their base
  - Thus binary search is “about  $\log N$ ” and Mergesort is “about  $N \log N$ ”
- You will often see these approximations written as  $O(N^2)$  or  $O(N \log N)$ , which we will define more specifically later

# Why Approximate?

## Reason 1: Architectures Vary

- It may seem strange that we treat  $2N$  as roughly the same running time as  $N$  or  $N/2$ . Why ignore these “constant factors” of 2 or  $1/2$ ?
- Reason #1: **Our analysis can't be that precise because of differences in machines and languages.** We're just counting “operations” but that lumps together addition, multiplication, requesting memory ... lots of stuff that all takes a different amount of time. That means there are really a bunch of unknown constants we don't know lurking in our analysis:  $a$  the time to add,  $m$  the time to request more memory...

# Why Approximate?

## Reason 2: We Care About the Long Run

- It may seem strange that we treat  $2N$  as roughly the same running time as  $N$  or  $N/2$ . Why ignore these “constant factors” of 2 or  $1/2$ ?
- Reason #2: **The constants don't matter in the long run.** Comparing two linear functions like  $N$  or  $2N$ , you might see a practical difference. But comparing different orders like  $2N$  vs  $N^2/2$  ... the one with  $N^2$  will always be worse *if there's enough data*. We'll explore this more when we talk about running times and big-O.