

# CSE 373: Data Structure & Algorithms

## Comparison Sorting

Riley Porter  
Winter 2017

# Course Logistics

- HW4 preliminary scripts out
- HW5 out → more graphs!
- Last main course topic this week: Sorting!
- Final exam in 2 weeks!

# Introduction to Sorting

- Why study sorting?
  - It uses information theory and is good algorithm practice!
- Different sorting algorithms have different trade-offs
  - No single “best” sort for all scenarios
  - Knowing one way to sort just isn’t enough
- Not usually asked about on tech interviews...
  - but if it comes up, you look bad if you can’t talk about it

# More Reasons to Sort

General technique in computing:

***Preprocess data to make subsequent operations faster***

Example: Sort the data so that you can

- Find the  $k^{\text{th}}$  largest in constant time for any  $k$
- Perform binary search to find elements in logarithmic time

Whether the performance of the preprocessing matters depends on

- How often the data will change (and how much it will change)
- How much data there is

# Definition: Comparison Sort

A computational problem with the following input and output

## **Input:**

An array **A** of length  $n$  comparable elements

## **Output:**

The same array **A**, containing the same elements where:

for any  $i$  and  $j$  where  $0 \leq i < j < n$   
then  $\mathbf{A}[i] \leq \mathbf{A}[j]$

# More Definitions

## **In-Place Sort:**

A sorting algorithm is in-place if it requires only  $O(1)$  extra space to sort the array.

- Usually modifies input array
- Can be useful: lets us minimize memory

## **Stable Sort:**

A sorting algorithm is stable if any equal items remain in the same relative order before and after the sort.

- Items that 'compare' the same might not be exact duplicates
- Might want to sort on some, but not all attributes of an item
- Can be useful to sort on one attribute first, then another one

# Stable Sort Example

## Input:

```
[(8, "fox"), (9, "dog"), (4, "wolf"), (8, "cow")]
```

Compare function: compare pairs by number only

## Output (stable sort):

```
[(4, "wolf"), (8, "fox"), (8, "cow"), (9, "dog")]
```

## Output (unstable sort):

```
[(4, "wolf"), (8, "cow"), (8, "fox"), (9, "dog")]
```

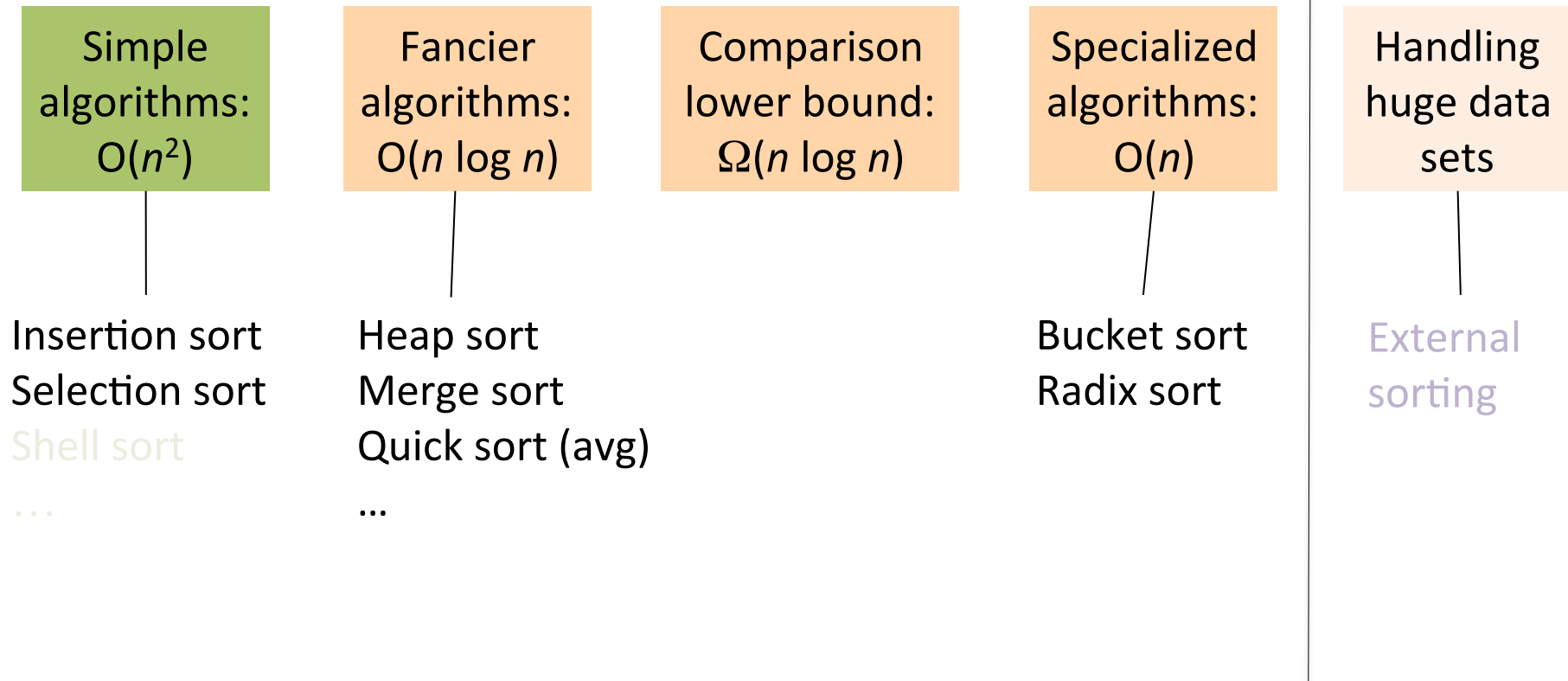
# Lots of algorithms for sorting...

Quicksort, Merge sort, In-place merge sort, Heap sort, Insertion sort, Intro sort, Selection sort, Timsort, Cubesort, Shell sort, Bubble sort, Binary tree sort, Cycle sort, Library sort, Patience sorting, Smoothsort, Strand sort, Tournament sort, Cocktail sort, Comb sort, Gnome sort, Block sort, Stackoverflow sort, Odd-even sort, Pigeonhole sort, Bucket sort, Counting sort, Radix sort, Spreadsort, Burstsor, Flashsort, Postman sort, Bead sort, Simple pancake sort, Spaghetti sort, Sorting network, Bitonic sort, Bogosort, Stooge sort, Insertion sort, Slow sort, Rainbow sort...

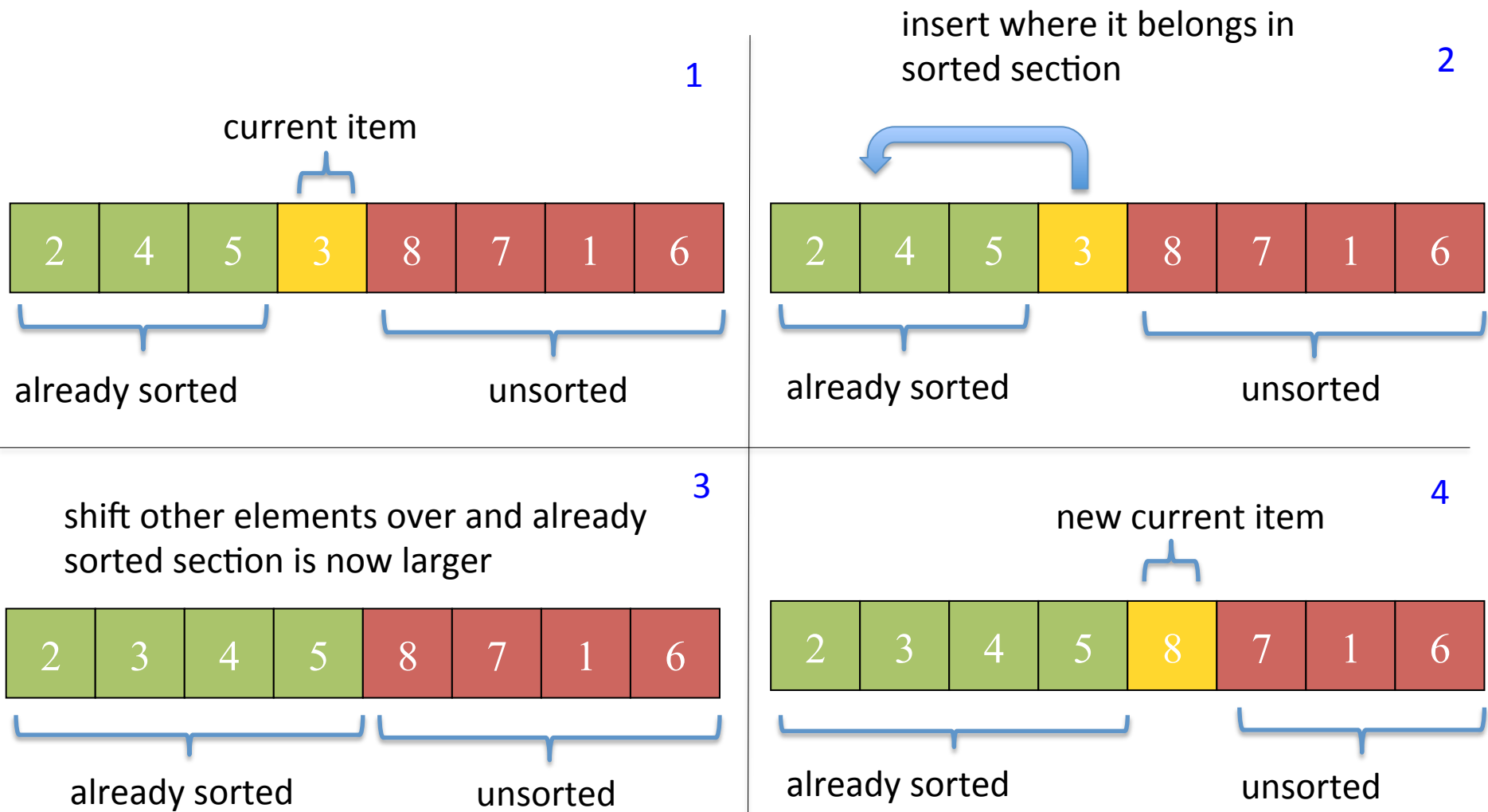
```
DEFINE FASTBOGOSORT(LIST):  
  // AN OPTIMIZED BOGOSORT  
  // RUNS IN  $O(N \log N)$   
  FOR N FROM 1 TO LOG(LENGTH(LIST)):  
    SHUFFLE(LIST):  
    IF ISSORTED(LIST):  
      RETURN LIST  
  RETURN "KERNEL PAGE FAULT (ERROR CODE: 2)"
```



# Sorting: The Big Picture



# Insertion Sort



# Insertion Sort

- Idea: At step  $k$ , put the  $k^{\text{th}}$  element in the correct position among the first  $k$  elements

```
for (int i = 0; i < n; i++) {  
    // Find index to insert into  
    int newIndex = findPlace(i);  
    // Insert and shift nodes over  
    shift(newIndex, i);  
}
```

- **Loop invariant:** when loop index is  $i$ , first  $i$  elements are sorted
- Runtime?  
Best-case \_\_\_\_\_ Worst-case \_\_\_\_\_ Average-case \_\_\_\_\_
- Stable? \_\_\_\_\_ In-place? \_\_\_\_\_

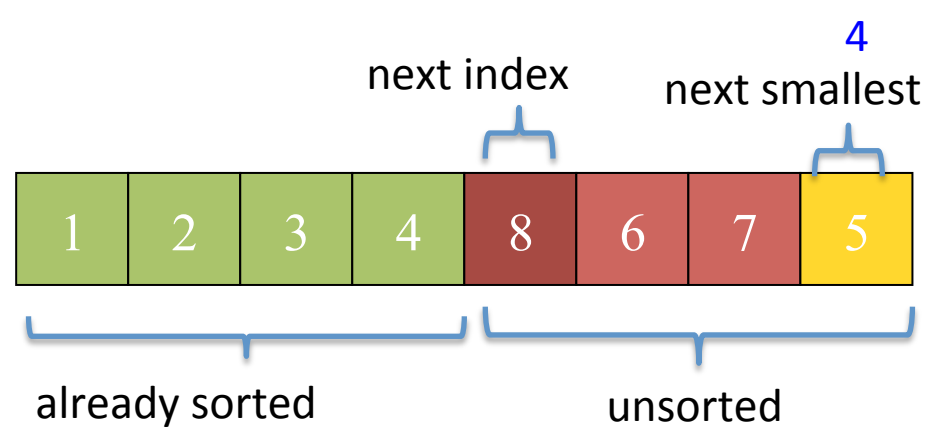
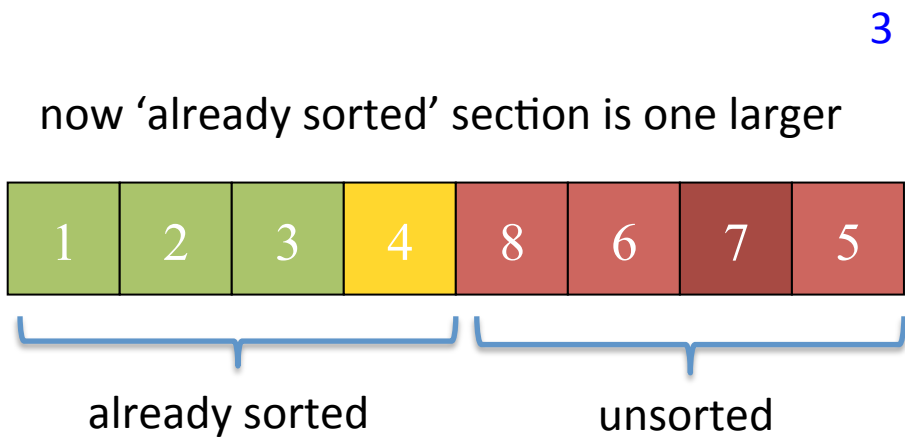
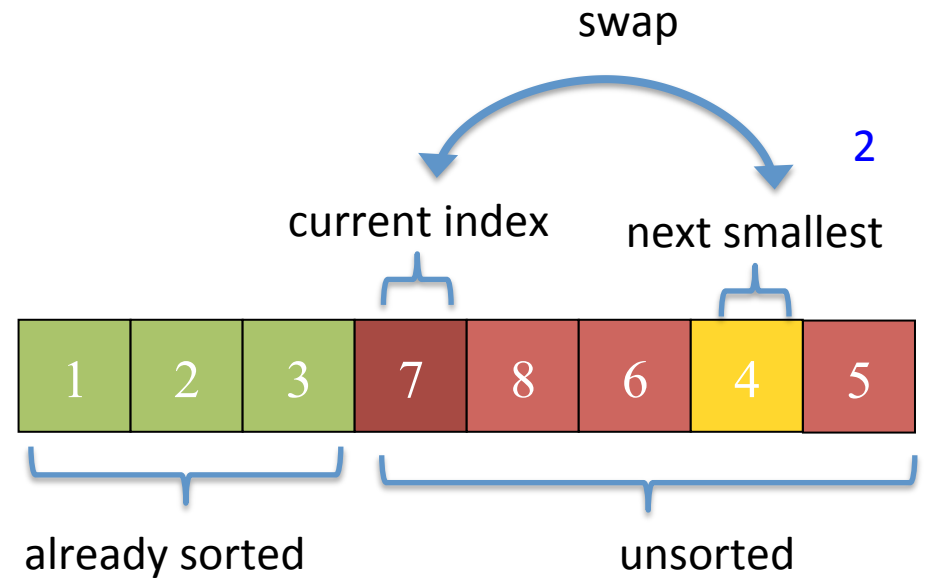
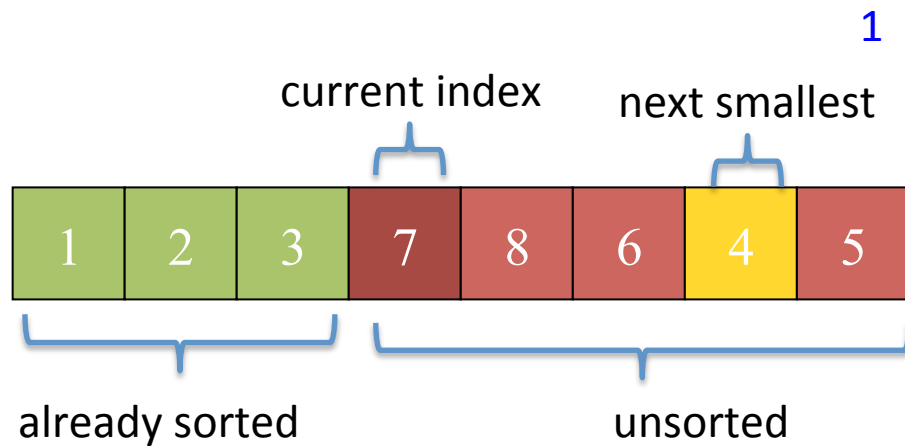
# Insertion Sort

- Idea: At step  $k$ , put the  $k^{\text{th}}$  element in the correct position among the first  $k$  elements

```
for (int i = 0; i < n; i++) {  
    // Find index to insert into  
    int newIndex = findPlace(i);  
    // Insert and shift nodes over  
    shift(newIndex, i);  
}
```

- **Loop invariant:** when loop index is  $i$ , first  $i$  elements are sorted
- Runtime?  
Best-case  $O(n)$  Worst-case  $O(n^2)$  Average-case  $O(n^2)$   
start sorted start reverse sorted (see text)
- Stable? Depends on implementation. Usually. In-place? Yes

# Selection Sort



# Selection Sort

- Idea: At step  $k$ , find the smallest element among the not-yet-sorted elements and put it at position  $k$

```
for (int i = 0; i < n; i++) {  
    // Find next smallest  
    int newIndex = findNextMin(i);  
    // Swap current and next smallest  
    swap(newIndex, i);  
}
```

- **Loop invariant:** when loop index is  $i$ , first  $i$  elements are sorted
- Runtime?  
Best-case \_\_\_\_\_ Worst-case \_\_\_\_\_ Average-case \_\_\_\_\_
- Stable? \_\_\_\_\_ In-place? \_\_\_\_\_

# Selection Sort

- Idea: At step  $k$ , find the smallest element among the not-yet-sorted elements and put it at position  $k$

```
for (int i = 0; i < n; i++) {  
    // Find next smallest  
    int newIndex = findNextMin(i);  
    // Swap current and next smallest  
    swap(newIndex, i);  
}
```

- **Loop invariant:** when loop index is  $i$ , first  $i$  elements are sorted
- Runtime?  
Best-case, Worst-case, and Average-case  $O(n^2)$
- Stable? **Depends on implementation. Usually.** In-place? **Yes**

# Insertion Sort vs. Selection Sort

- Have the same worst-case and average-case asymptotic complexity
  - Insertion-sort has better best-case complexity; preferable when input is “mostly sorted”
- Useful for small arrays or for mostly sorted input

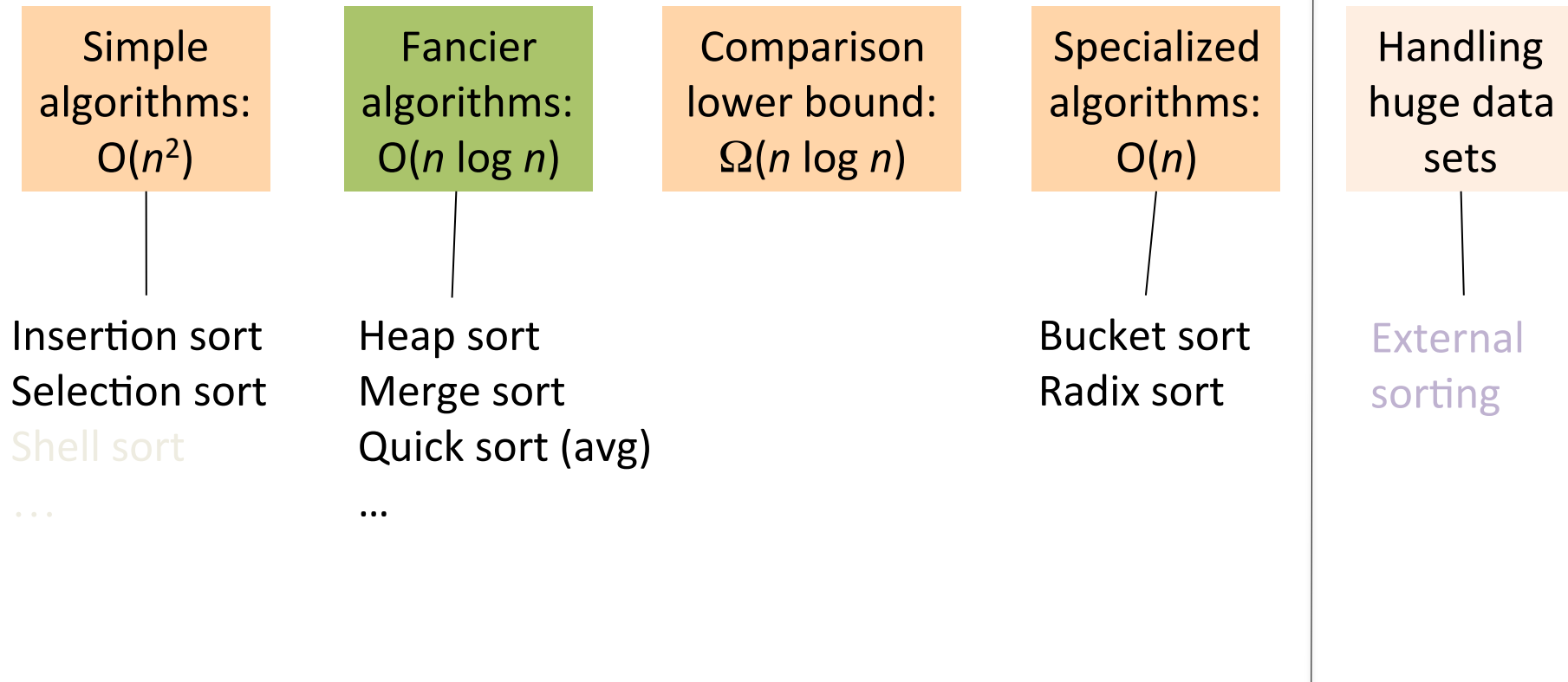


# Bubble Sort

- for  $n$  iterations: 'bubble' next largest element to the end of the unsorted section, by doing a series of swaps
- Not intuitive – It's unlikely that you'd come up with bubble sort
- Not good asymptotic complexity:  $O(n^2)$
- It's not particularly efficient with respect to common factors

Basically, almost never is better than insertion or selection sort.

# Sorting: The Big Picture



# Heap Sort

- Idea: buildHeap then call deleteMin  $n$  times

```
E[] input = buildHeap(...);
E[] output = new E[n];
for (int i = 0; i < n; i++) {
    output[i] = deleteMin(input);
}
```

- Runtime?  
Best-case \_\_\_\_ Worst-case \_\_\_\_ Average-case \_\_\_\_
- Stable? \_\_\_\_\_
- In-place? \_\_\_\_\_

# Heap Sort

- Idea: `buildHeap` then call `deleteMin`  $n$  times

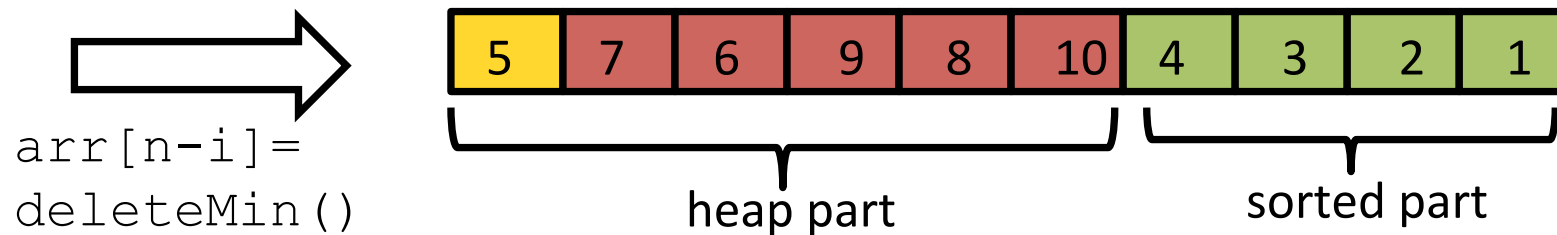
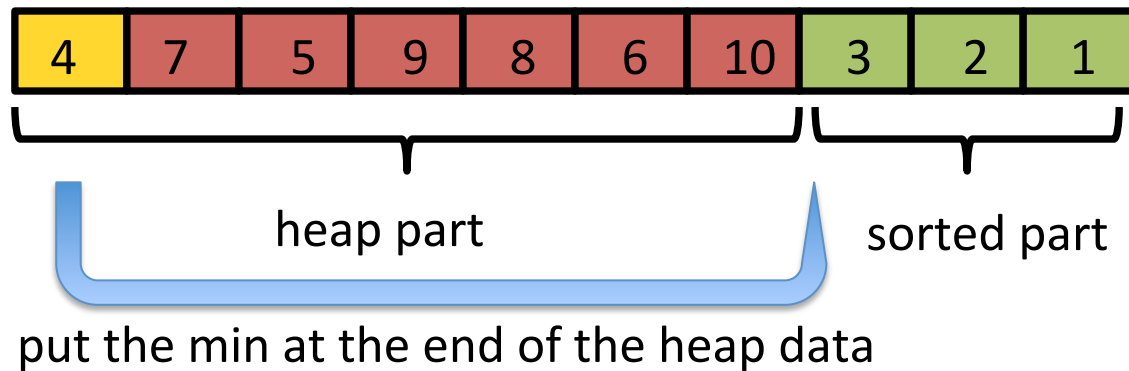
```
E[] input = buildHeap(...);
E[] output = new E[n];
for (int i = 0; i < n; i++) {
    output[i] = deleteMin(input);
}
```

- Runtime?  
Best-case, Worst-case, and Average-case:  $O(n \log(n))$
- Stable? **No**
- In-place? **No. But it could be, with a slight trick...**

# In-place Heap Sort

But this reverse sorts –  
how would you fix that?

- Treat the initial array as a heap (via **buildHeap**)
- When you delete the  $i^{\text{th}}$  element, put it at **arr[n-i]**
  - That array location isn't needed for the heap anymore!



# “AVL sort”? “Hash sort”?

**AVL Tree:** sure, we can also use an AVL tree to:

- **insert** each element: total time  $O(n \log n)$
- Repeatedly **deleteMin**: total time  $O(n \log n)$ 
  - Better: in-order traversal  $O(n)$ , but still  $O(n \log n)$  overall
- But this cannot be done in-place and has worse constant factors than heap sort

**Hash Structure:** don't even think about trying to sort with a hash table!

- Finding min item in a hashtable is  $O(n)$ , so this would be a slower, more complicated selection sort

# Divide and conquer

Divide-and-conquer is a useful technique for solving many kinds of problems (not just sorting). It consists of the following steps:

1. Divide your work up into smaller pieces (recursively)
2. Conquer the individual pieces (as base cases)
3. Combine the results together (recursively)

```
algorithm(input) {  
  if (small enough) {  
    CONQUER, solve, and return input  
  } else {  
    DIVIDE input into multiple pieces  
    RECURSE on each piece  
    COMBINE and return results  
  }  
}
```

# Divide-and-Conquer Sorting

Two great sorting methods are fundamentally divide-and-conquer

## **Mergesort:**

- Sort the left half of the elements (recursively)

- Sort the right half of the elements (recursively)

- Merge the two sorted halves into a sorted whole

## **Quicksort:**

- Pick a “pivot” element

- Divide elements into less-than pivot and greater-than pivot

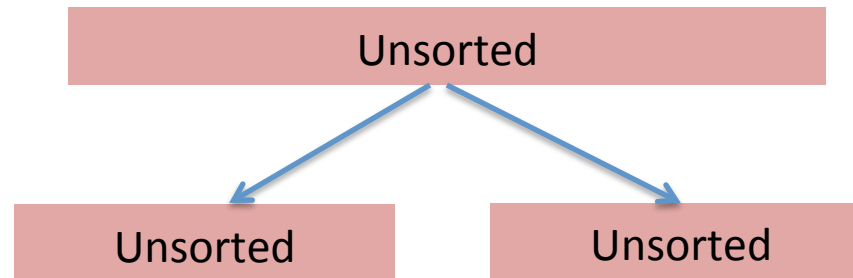
- Sort the two divisions (recursively on each)

- Answer is: sorted-less-than....pivot....sorted-greater-than



# Merge Sort

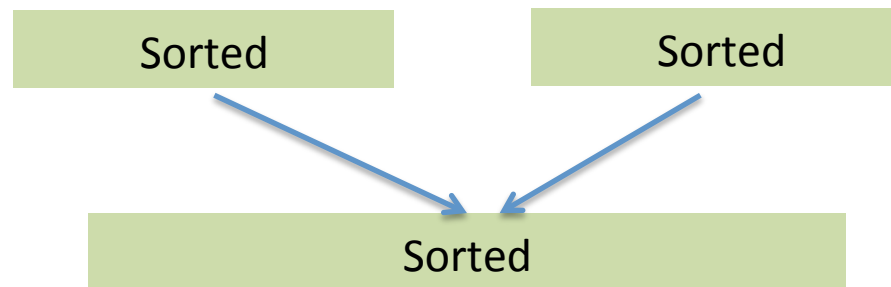
**Divide:** Split array roughly into half



**Conquer:** Return array when length  $\leq 1$



**Combine:** Combine two sorted arrays using merge

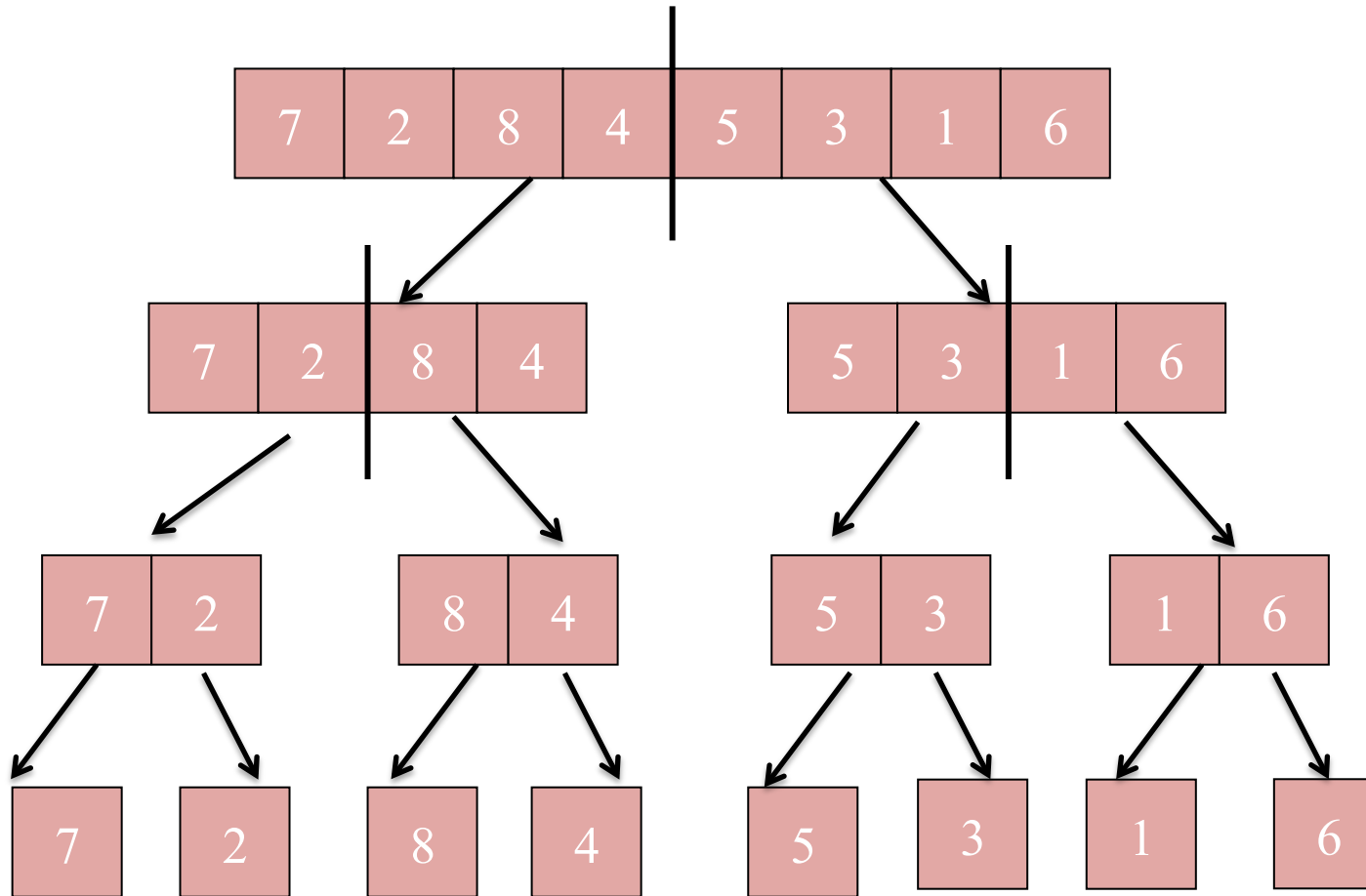


# Merge Sort: Pseudocode

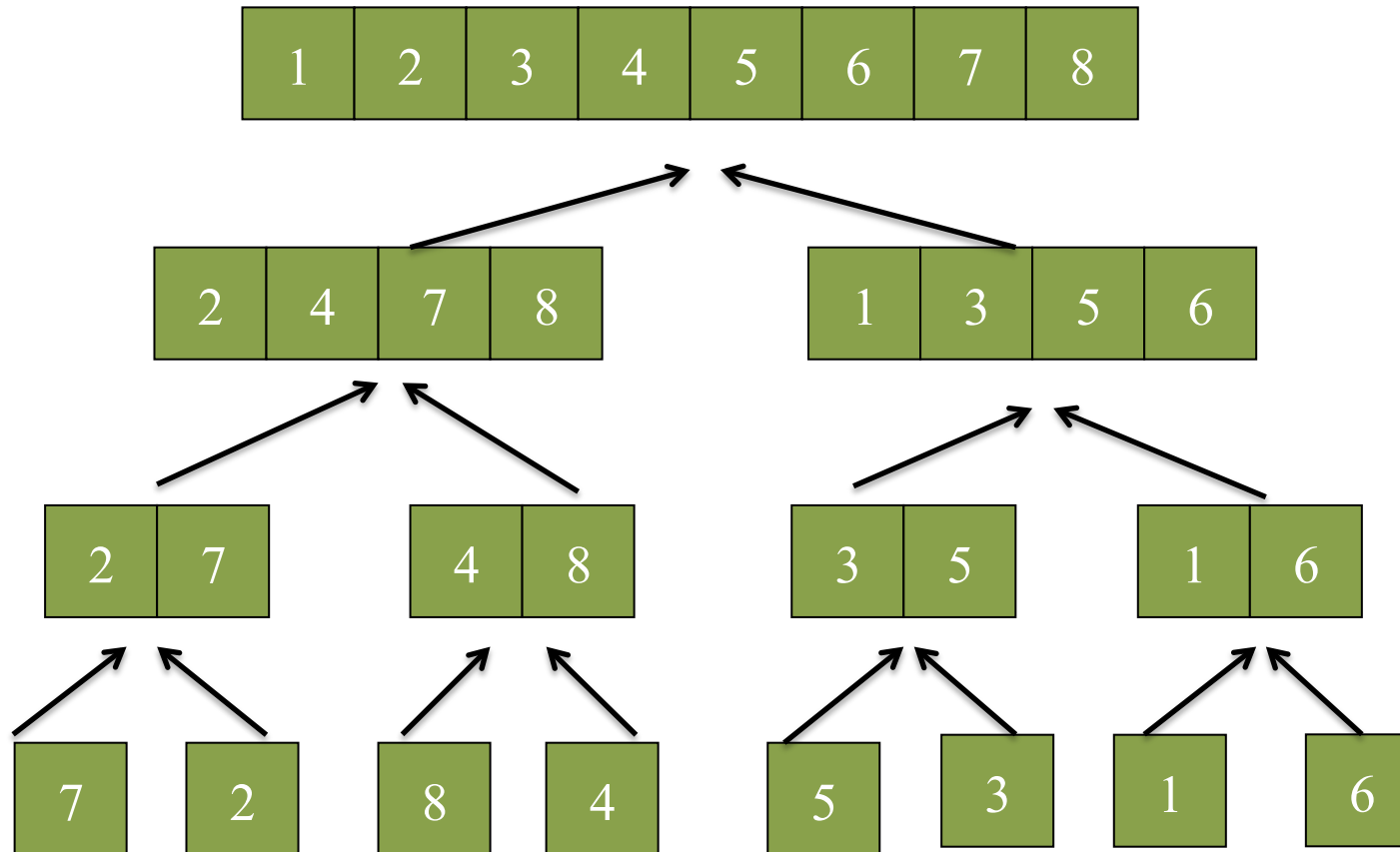
Core idea: split array in half, sort each half, merge back together. If the array has size 0 or 1, just return it unchanged

```
mergesort(input) {  
    if (input.length < 2) {  
        return input;  
    } else {  
        smallerHalf = sort(input[0, ..., mid]);  
        largerHalf = sort(input[mid + 1, ...]);  
        return merge(smallerHalf, largerHalf);  
    }  
}
```

# Merge Sort Example



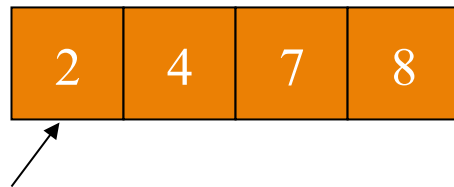
# Merge Sort Example



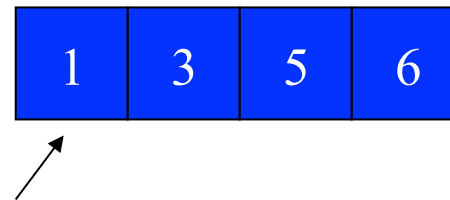
# Merge Example

**Merge operation:** Use 3 pointers and 1 more array

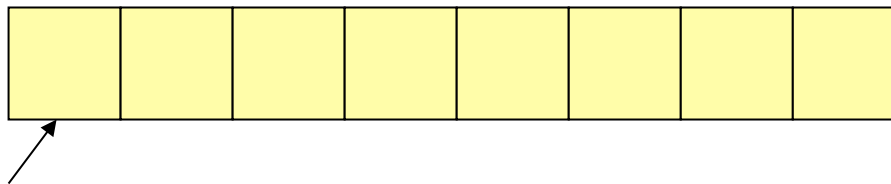
First half after sort:



Second half after sort:



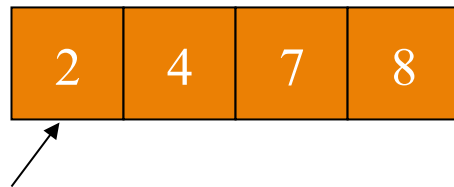
Result:



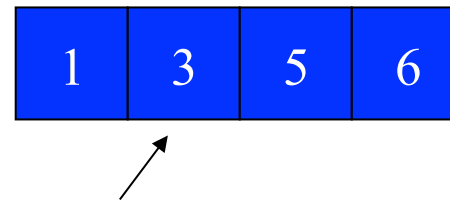
# Merge Example

**Merge operation:** Use 3 pointers and 1 more array

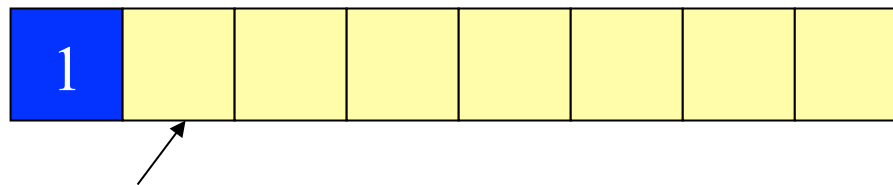
First half after sort:



Second half after sort:



Result:



# Merge Example

**Merge operation:** Use 3 pointers and 1 more array

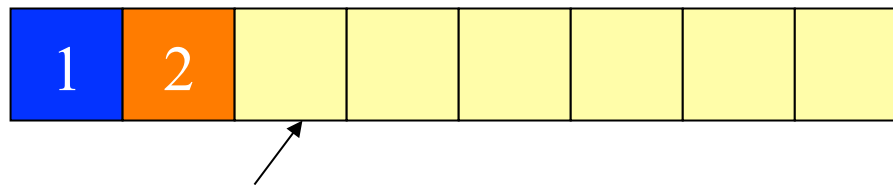
First half after sort:



Second half after sort:



Result:



# Merge Example

**Merge operation:** Use 3 pointers and 1 more array

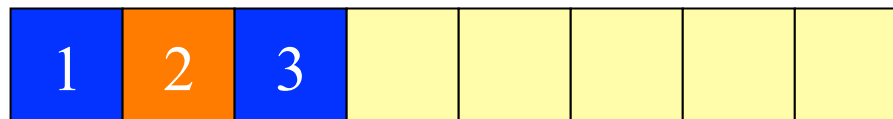
First half after sort:



Second half after sort:



Result:





# Merge Example

**Merge operation:** Use 3 pointers and 1 more array

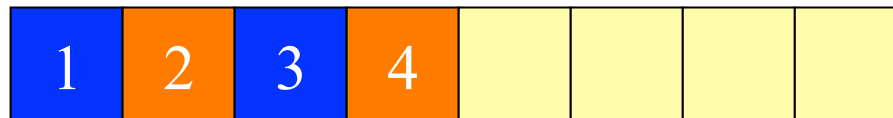
First half after sort:



Second half after sort:



Result:



# Merge Example

**Merge operation:** Use 3 pointers and 1 more array

First half after sort:



Second half after sort:



Result:



# Merge Example

**Merge operation:** Use 3 pointers and 1 more array

First half after sort:



Second half after sort:



Result:



# Merge Example

**Merge operation:** Use 3 pointers and 1 more array

First half after sort:



Second half after sort:



Result:



# Merge Example

**Merge operation:** Use 3 pointers and 1 more array

First half after sort:



Second half after sort:



Result:



**After Merge:** copy result into original unsorted array.

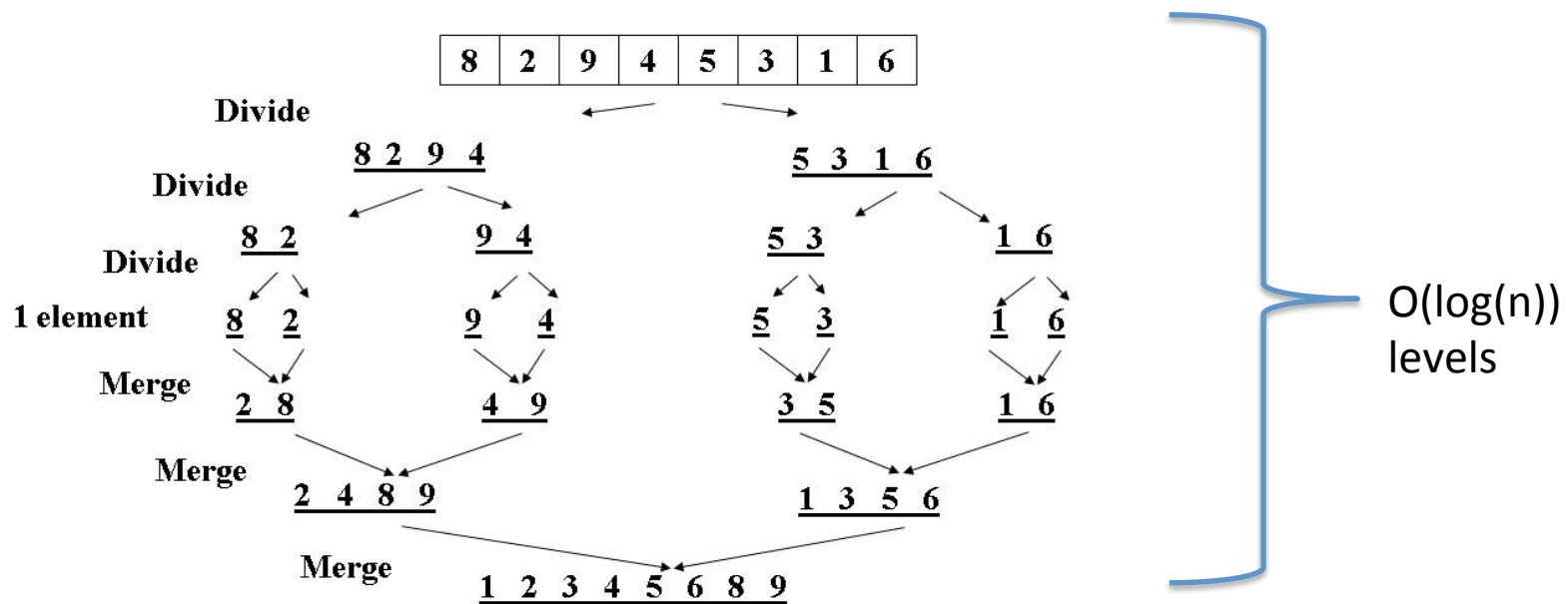
Or you can do the whole process in-place, but it's more difficult to write

# Merge Sort Analysis

Runtime:

- subdivide the array in half each time:  $O(\log(n))$  recursive calls
- merge is an  $O(n)$  traversal at each level

So, the best and worst case runtime is the same:  $O(n \log(n))$



# Merge Sort Analysis

## **Stable?**

Yes! If we implement the merge function correctly, merge sort will be stable.

## **In-place?**

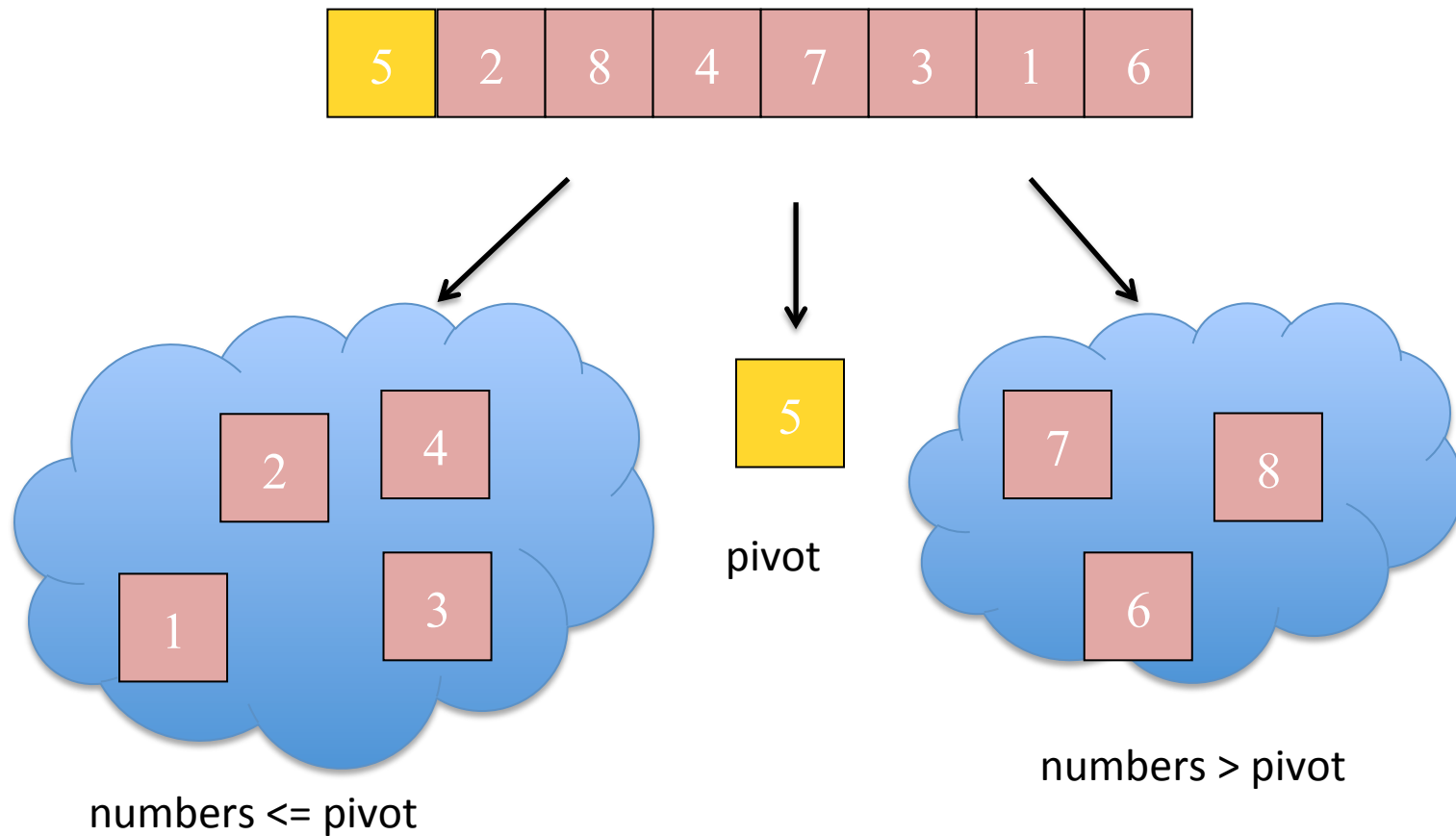
No. Unless you want to give yourself a headache. Merge must construct a new array to contain the output, so merge sort is not in-place.

We're constantly copying and creating new arrays at each level...

**One Solution:** (less of a headache than actually implementing in-place) create a single auxiliary array and swap between it and the original on each level.

# Quick Sort

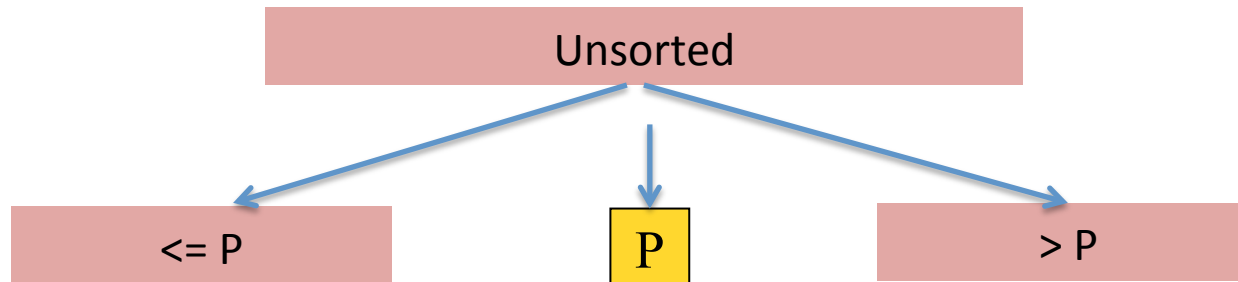
**Divide:** Split array around a 'pivot'





# Quick Sort

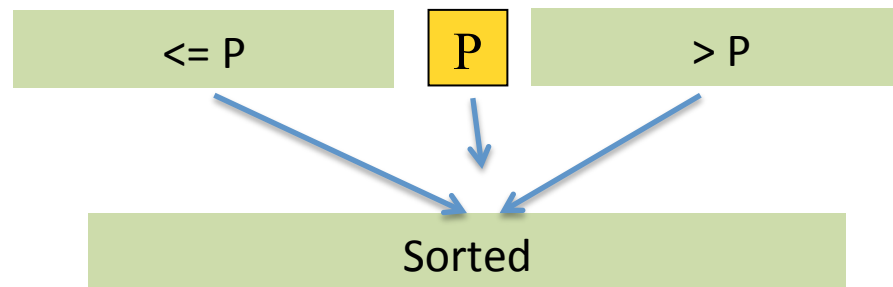
**Divide:** Pick a pivot, partition into groups



**Conquer:** Return array when length  $\leq 1$



**Combine:** Combine sorted partitions and pivot

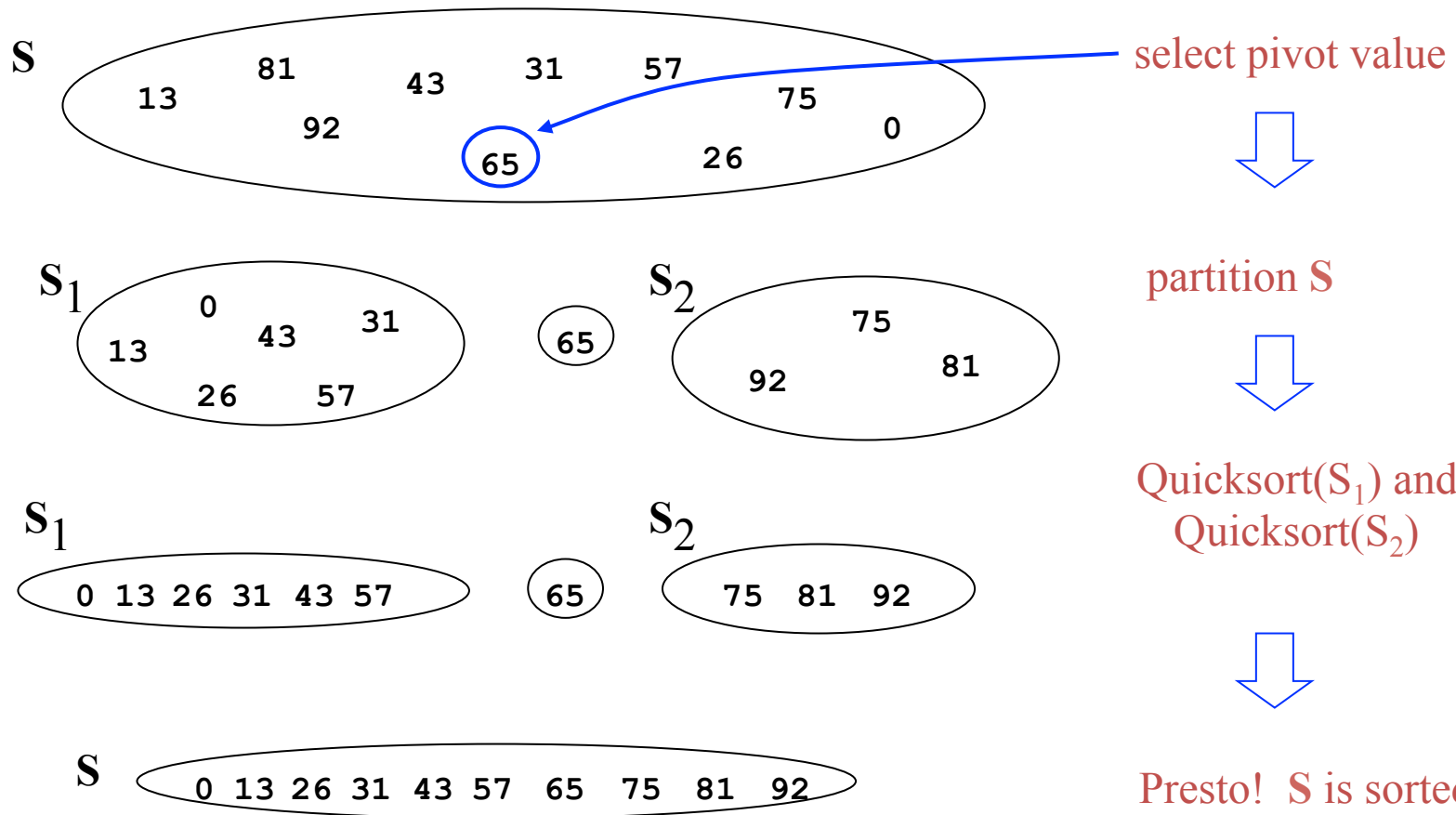


# Quick Sort Pseudocode

Core idea: Pick some item from the array and call it the pivot. Put all items smaller in the pivot into one group and all items larger in the other and recursively sort. If the array has size 0 or 1, just return it unchanged.

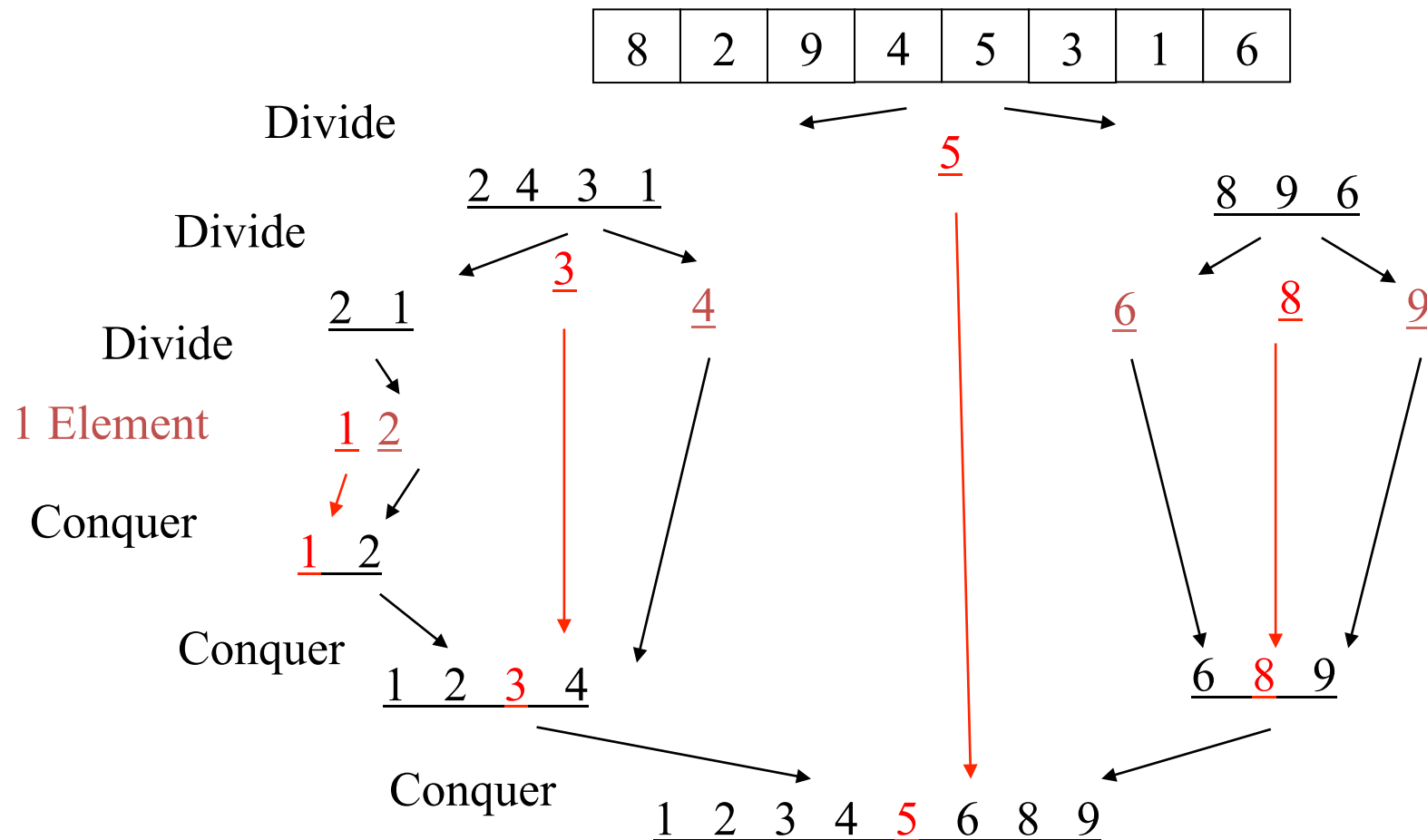
```
quicksort(input) {
  if (input.length < 2) {
    return input;
  } else {
    pivot = getPivot(input);
    smallerHalf = sort(getSmaller(pivot, input));
    largerHalf = sort(getBigger(pivot, input));
    return smallerHalf + pivot + largerHalf;
  }
}
```

# Think in Terms of Sets



[Weiss]

# Example, Showing Recursion



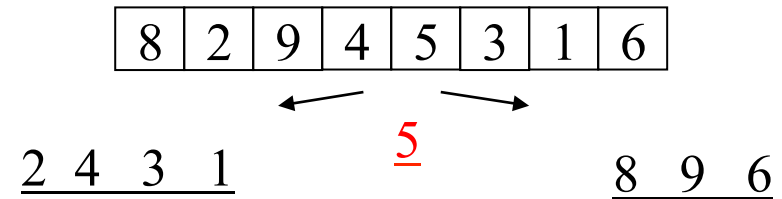
# Details

Have not yet explained:

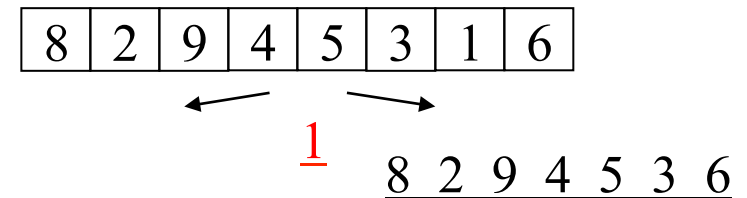
- How to pick the pivot element
  - Any choice is correct: data will end up sorted
  - But as analysis will show, want the two partitions to be about equal in size
- How to implement partitioning
  - In linear time
  - In place

# Pivots

- Best pivot?
  - Median
  - Halve each time



- Worst pivot?
  - Greatest/least element
  - Problem of size  $n - 1$
  - $O(n^2)$



# Potential pivot rules

While sorting **arr** from **lo** (inclusive) to **hi** (exclusive)...

- Pick **arr[lo]** or **arr[hi-1]**
  - Fast, but worst-case occurs with mostly sorted input
- Pick random element in the range
  - Does as well as any technique, but (pseudo)random number generation can be slow
  - Still probably the most elegant approach
- Median of 3, e.g., **arr[lo]** , **arr[hi-1]** , **arr[(hi+lo)/2]**
  - Common heuristic that tends to work well

# Partitioning

- Conceptually simple, but hardest part to code up correctly
  - After picking pivot, need to partition in linear time in place
- One approach (there are slightly fancier ones):
  1. Swap pivot with `arr[lo]`
  2. Use two fingers `i` and `j`, starting at `lo+1` and `hi-1`
  3. `while (i < j)`
    - `if (arr[j] > pivot) j--`
    - `else if (arr[i] < pivot) i++`
    - `else swap arr[i] with arr[j]`
  4. Swap pivot with `arr[i]` \*

\*skip step 4 if pivot ends up being least element



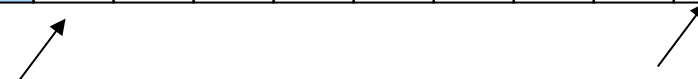
# Example

- **Step one:** pick pivot as median of 3
  - $lo = 0$ ,  $hi = 10$

0	1	2	3	4	5	6	7	8	9
<b>8</b>	1	4	9	<b>0</b>	3	5	2	7	<b>6</b>

- **Step two:** move pivot to the  $lo$  position

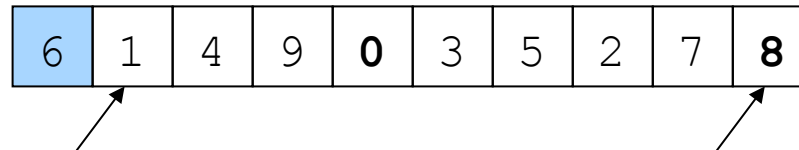
0	1	2	3	4	5	6	7	8	9
<b>6</b>	1	4	9	<b>0</b>	3	5	2	7	<b>8</b>



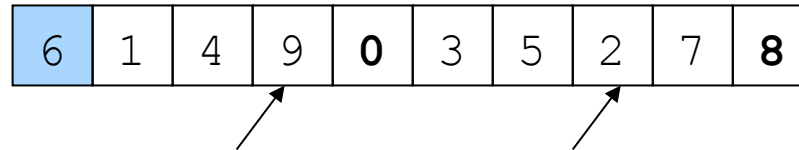
# Example

Often have more than one swap during partition – this is a short example

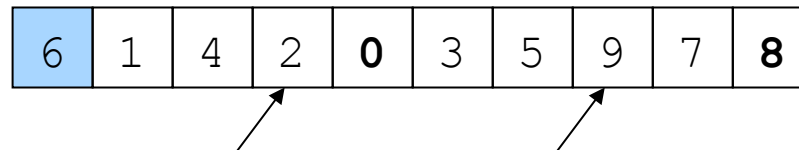
Now partition in place



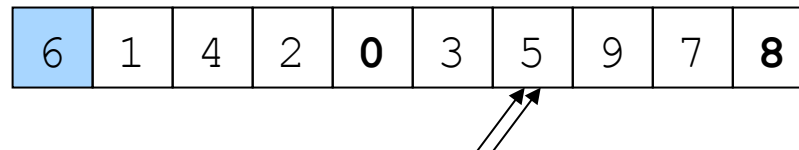
Move fingers



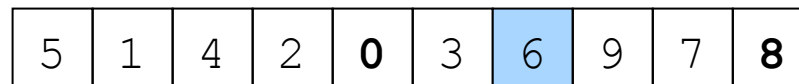
Swap



Move fingers



Move pivot



# Analysis

- **Best-case:** Pivot is always the median  
 $T(0)=T(1)=1$   
 $T(n)=2T(n/2) + n$       -- linear-time partition  
Same recurrence as mergesort:  $O(n \log n)$
- **Worst-case:** Pivot is always smallest or largest element  
 $T(0)=T(1)=1$   
 $T(n) = 1T(n-1) + n$   
Basically same recurrence as selection sort:  $O(n^2)$
- **Average-case** (e.g., with random pivot)
  - $O(n \log n)$ , not responsible for proof (in text)

# Cutoffs

- For small  $n$ , all that recursion tends to cost more than doing a quadratic sort
  - Remember asymptotic complexity is for large  $n$
- Common engineering technique: switch algorithm below a **cutoff**
  - Reasonable rule of thumb: use insertion sort for  $n < 10$
- Notes:
  - Could also use a cutoff for merge sort
  - Cutoffs are also the norm with parallel algorithms
    - Switch to sequential algorithm
  - None of this affects asymptotic complexity

# Cutoff Pseudocode

```
void quicksort(int[] arr, int lo, int hi) {  
    if (hi - lo < CUTOFF)  
        insertionSort(arr, lo, hi);  
    else  
        ...  
}
```

Notice how this cuts out the vast majority of the recursive calls

- Think of the recursive calls to quicksort as a tree
- Trims out the bottom layers of the tree

# Today's Takeaways

- Understand how basic sorting works:
  - insertion sort
  - selection sort
  - bubble sort
- Understand how  $n \log(n)$  sorting works:
  - heap sort
  - merge sort
  - quick sort
- Cool links:
  - <http://www.sorting-algorithms.com/>
  - <https://www.youtube.com/watch?v=t8g-iYGHpEA>