

# Lecture 11: Comparison Sorts

CSE 332: Data Structures & Parallelism

Yafqa Khan

Summer 2025

# Announcements

- EX04: AVL
  - Due Next Monday
- Exam 1 next Friday

# Today

- **Sorting Algorithm 1: Insertion Sort**
- Sorting Algorithm 2: Selection Sort
- Sorting Algorithm 3: Heap Sort
  - In-place optimization
- Sorting Algorithm 4: Merge Sort
  - Merging
- Sorting Algorithm 5: Quick Sort
  - Picking a pivot
  - Partitioning
- Comparison Sorting Lower Bound

# Sorting: An introduction

- Why sorting?
  - Want to know "all the data items" in some order
  - Very common to need data sorted somehow
    - Alphabetical list of people
    - Population list of countries
    - Search engine results by relevance
    - Binary search
- Why many ways of sorting?
  - Tradeoffs...
    - Asymptotic vs Constant Factors
    - Different properties

# Sorting: Goals (Terminology)

## 1. Stable

- Maybe in the case of ties we should preserve the original ordering
- One way to sort twice, Ex: Sort movies by year, then for ties, alphabetically

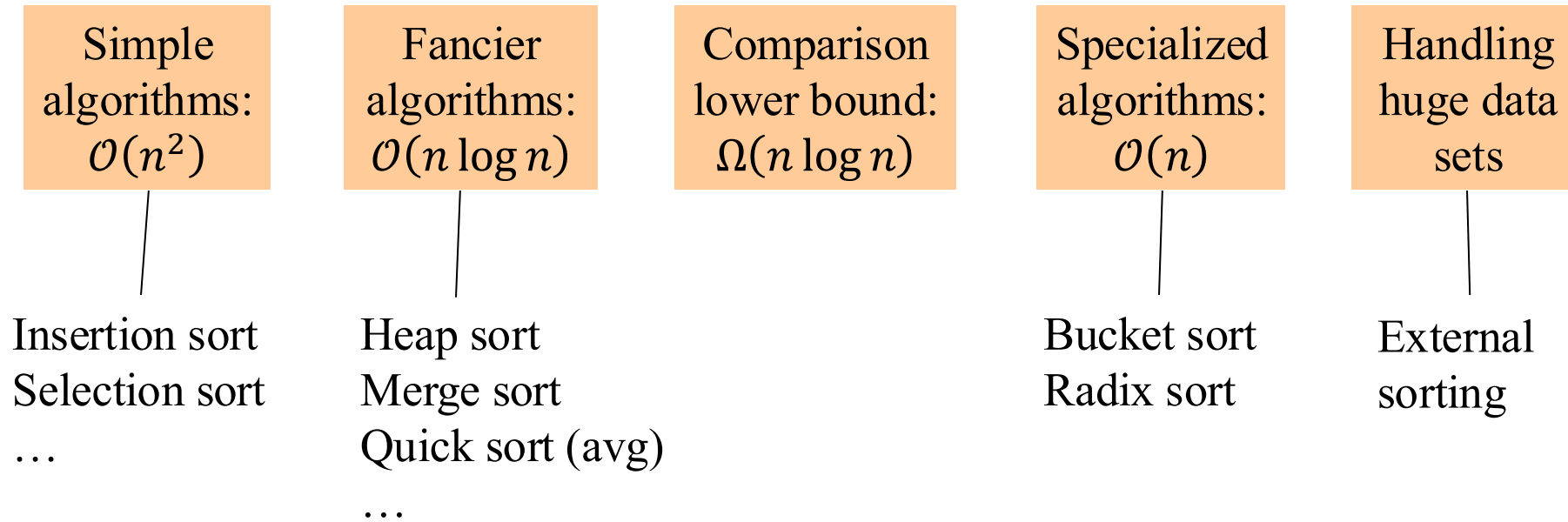
## 2. In-Place (Space)

- No more than  $\mathcal{O}(1)$  "auxiliary space"
- Only use original array by swapping elements

## 3. Fast (Time)

- Typically,  $\mathcal{O}(n \log n)$
- Or good constant factors

# Sorting: The Big Picture



# Sorting Algorithm 1: Insertion Sort

Intuition: Given a hand of cards, sort it

Algorithm:

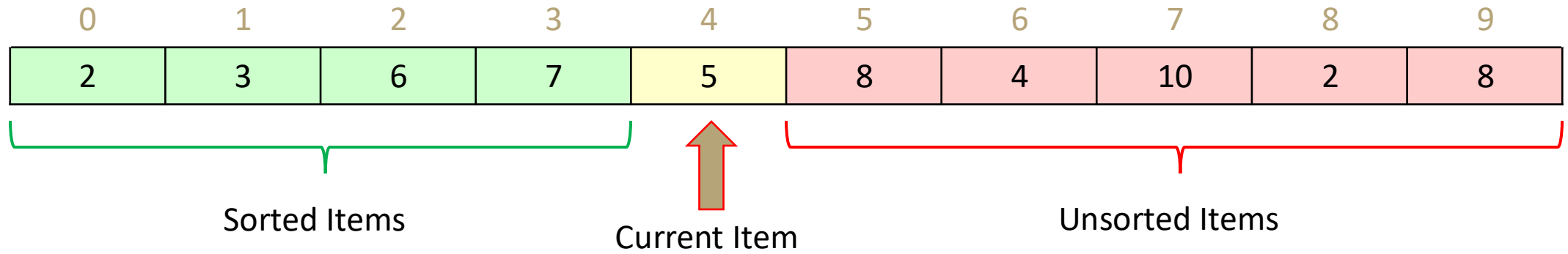
- Maintain a sorted subarray
  1. Sort first 2 elements
  2. Insert 3rd element in order
  3. Insert 4th element in order
  4. ...

# Insertion Sort: Pseudocode

```
insertionSort(int[] arr){  
    for(i=0; i < arr.length; i++){  
        int curr = i  
        while(arr[curr-1] > arr[curr]){  
            swap(arr[curr-1], arr[curr])  
            curr -= 1  
        }  
    }  
}
```



# Insertion Sort: Visual



# Insertion Sort: Analysis

## 1. Stable?

- Yes!

## 2. In-Place?

- Yes!

## 3. Fast?

- No :( (in terms of asymptotics)
  - Best Case:  $\mathcal{O}(n)$
  - Worst Case:  $\mathcal{O}(n^2)$
- **Good constant factors!**

# Today

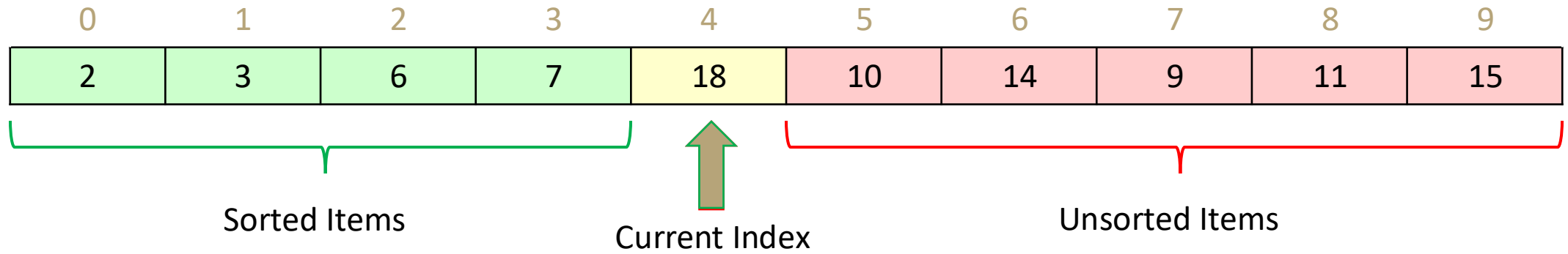
- Sorting Algorithm 1: Insertion Sort
- **Sorting Algorithm 2: Selection Sort**
- Sorting Algorithm 3: Heap Sort
  - In-place optimization
- Sorting Algorithm 4: Merge Sort
  - Merging
- Sorting Algorithm 5: Quick Sort
  - Picking a pivot
  - Partitioning
- Comparison Sorting Lower Bound

# Sorting Algorithm 2: Selection Sort

Algorithm:

- Maintain a sorted subarray
  1. Find the smallest element remaining in the unsorted subarray
  2. Append it at the end of the sorted part
  3. Repeat

# Selection Sort: Visual



# Selection Sort: Analysis

## 1. Stable?

- No :( (e.g., try  $[2_1, 2_2, 1]$ )

## 2. In-Place?

- Yes!

## 3. Fast?

- No :( (in terms of asymptotics)
  - Best Case:  $\mathcal{O}(n^2)$ 
    - Worse than insertion sort when array is almost fully sorted
  - Worst Case:  $\mathcal{O}(n^2)$
- **Good constant factors!**

# Sorting Algorithm `null`: Bubble Sort

- We pretend it doesn't exist
- Bad asymptotic complexity:  $\mathcal{O}(n^2)$
- Bad constant factors
- Literally should never be used
  - Anything it is good at, another algorithm is at least good at
- **IDK WHY THE INTERNET LIKES USING IT**

Any Questions?



# Today

- Sorting Algorithm 1: Insertion Sort
- Sorting Algorithm 2: Selection Sort
- Sorting Algorithm 3: Heap Sort
  - In-place optimization
- Sorting Algorithm 4: Merge Sort
  - Merging
- Sorting Algorithm 5: Quick Sort
  - Picking a pivot
  - Partitioning
- Comparison Sorting Lower Bound

# Sorting Algorithm 3: Heap Sort

Intuition: Use a heap

Algorithm:

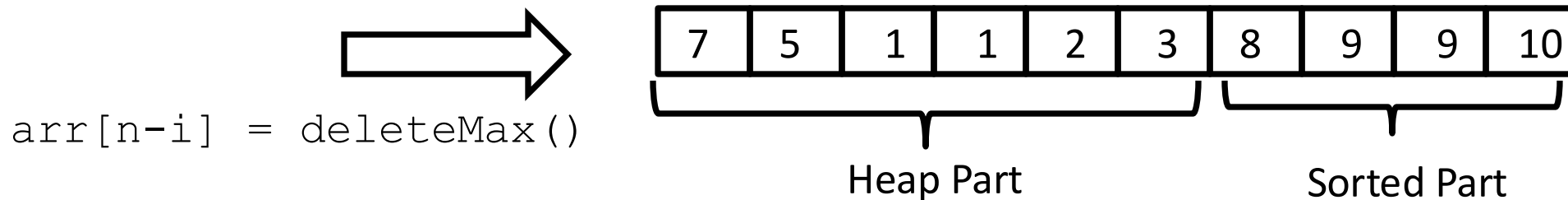
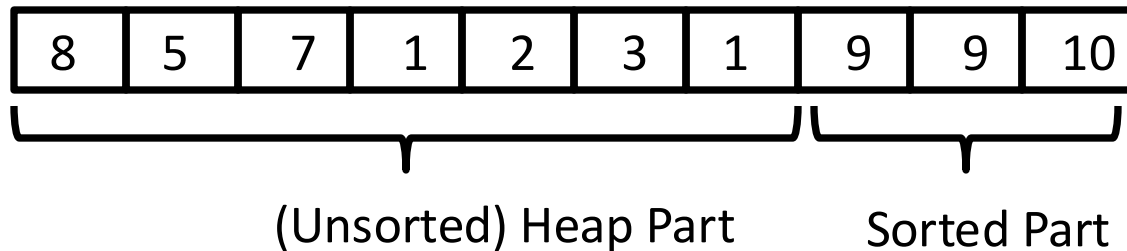
1. Put all elements into a heap (e.g., with `buildHeap`)
2. Remove elements one by one and put back into the array

# Heap Sort (unoptimized): Pseudocode

```
heapSort(int[] arr) {  
    heap = buildHeap(arr)  
    for(i=0; i < arr.length; i++) {  
        arr[i] = heap.deleteMin()  
    }  
}
```

# (Max) Heap Sort: In-place Optimization

- Treat the initial array as a heap (via `buildHeap`)
- When you delete the  $i$ th element, put it at `arr[n-i]` (the back)
  - It's not part of the heap anymore!



Any Questions?

# Heap Sort: Analysis

## 1. Stable?

No, no guarantees on which key comes first

- Technically it **can** be but it makes it not in-place (we don't talk about this).

## 2. In-Place?

- Yes!

## 3. Fast?

- Yes! (in terms of asymptotics)
  - Best Case:  $\mathcal{O}(n \log n)$
  - Worst Case:  $\mathcal{O}(n \log n)$
- **Worse constant factors...**
  - Think: have to maintain Heap, using `buildHeap`, etc.

# Sorting Algorithm `null`: AVL Sort

- We pretend it doesn't exist
- Idea  $\mathcal{O}(n \log n)$ :
  - insert all elements into some balanced tree,  $\mathcal{O}(n \log n)$
  - in-order traversal,  $\mathcal{O}(n)$
- Not in-place
- Worse constant factors
- Heap Sort is just better...

# Today

- Sorting Algorithm 1: Insertion Sort
- Sorting Algorithm 2: Selection Sort
- Sorting Algorithm 3: Heap Sort
  - In-place optimization
- **Sorting Algorithm 4: Merge Sort**
  - **Merging**
- Sorting Algorithm 5: Quick Sort
  - Picking a pivot
  - Partitioning
- Comparison Sorting Lower Bound



# Divide and Conquer



Very important technique in algorithm design

1. Divide problems into smaller parts
2. Solve each part independently
  - Think: recursion, parallelism (later)
3. Combine each part's solution to produce overall solution

e.g.,

- Sort each half of the array, combine together
- to sort each half, split into halves
- ...

# Divide and Conquer Sorting

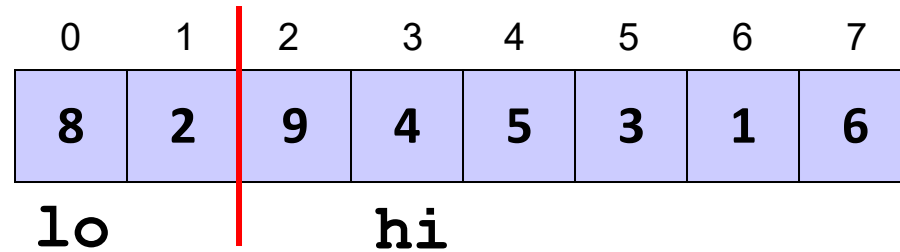
## 1. Merge Sort

- 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

## 2. Quick Sort

- Divide elements into those less-than pivot and those greater-than pivot
- Sort the two divisions (recursively on each)
- Merge as [**sorted-less-than** then **pivot** then **sorted-greater-than**]

# Sorting Algorithm 4: Merge Sort



- Algorithm, (recursively) sort from position `lo` to position `hi`:
  1. If `lo` to `hi` is 1 element long,
    1. Sorted! Because its 1 element...
  2. Else, split into halves:
    1. Sort from `lo` to  $(hi+lo) / 2$  (`lo` to the middle)
    2. Sort from  $(hi+lo) / 2$  to `hi`
    3. Merge the two halves together
- How to merge 2 **sorted** halves?
  - $\mathcal{O}(n)$  time but needs auxiliary space...

# Merge Sort: Merging Visualization

Start with:

8	2	9	4	5	3	1	6
---	---	---	---	---	---	---	---

After we return from  
left and right recursive calls  
(pretend it works for now)

2	4	8	9	1	3	5	6
---	---	---	---	---	---	---	---

Merge:

Use 3 pointers **aux**  
and 1 more array

--	--	--	--	--	--	--	--

(After merge,  
copy back to  
original array)

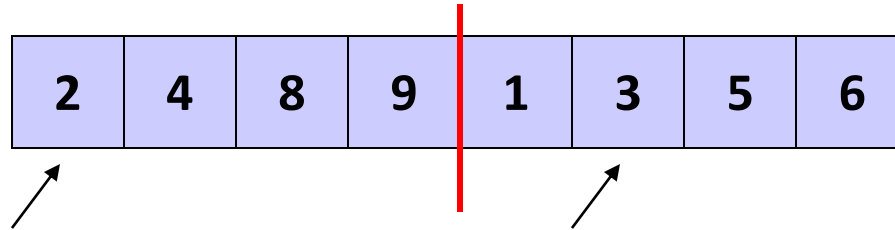
# Merge Sort: Merging Visualization (Soln.)

Start with:

8	2	9	4	5	3	1	6
---	---	---	---	---	---	---	---

After recursion:  
(not magic 😊)

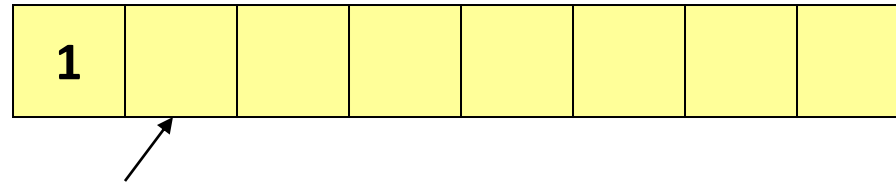
2	4	8	9	1	3	5	6
---	---	---	---	---	---	---	---



Merge:

Use 3 “fingers”  
and 1 more array

1							
---	--	--	--	--	--	--	--



(After merge,  
copy back to  
original array)

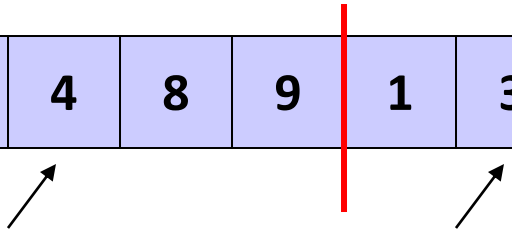
# Merge Sort: Merging Visualization (Soln.)

Start with:

8	2	9	4	5	3	1	6
---	---	---	---	---	---	---	---

After recursion:  
(not magic 😊)


2	4	8	9	1	3	5	6
---	---	---	---	---	---	---	---



Merge:

Use 3 “fingers”  
and 1 more array

1	2						
---	---	--	--	--	--	--	--



(After merge,  
copy back to  
original array)

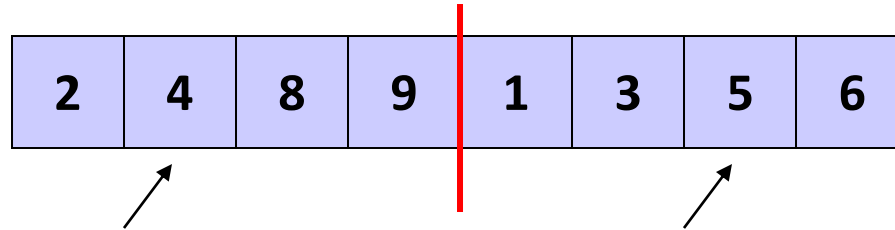
# Merge Sort: Merging Visualization (Soln.)

Start with:

8	2	9	4	5	3	1	6
---	---	---	---	---	---	---	---

After recursion:  
(not magic 😊)

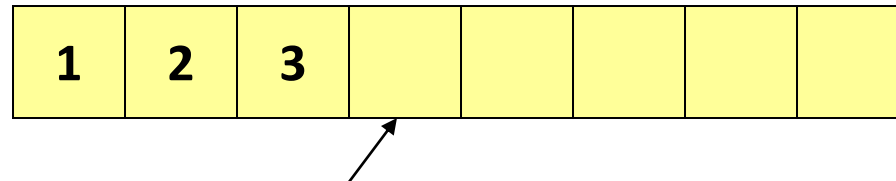
2	4	8	9	1	3	5	6
---	---	---	---	---	---	---	---



Merge:

Use 3 “fingers”  
and 1 more array

1	2	3					
---	---	---	--	--	--	--	--



(After merge,  
copy back to  
original array)

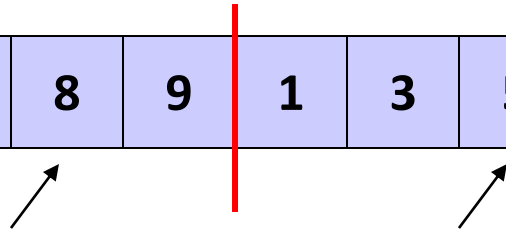
# Merge Sort: Merging Visualization (Soln.)

Start with:

8	2	9	4	5	3	1	6
---	---	---	---	---	---	---	---

After recursion:  
(not magic 😊)


2	4	8	9	1	3	5	6
---	---	---	---	---	---	---	---



Merge:

Use 3 “fingers”  
and 1 more array

1	2	3	4				
---	---	---	---	--	--	--	--



(After merge,  
copy back to  
original array)



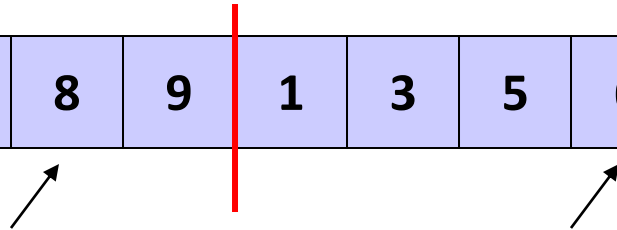
# Merge Sort: Merging Visualization (Soln.)

Start with:

8	2	9	4	5	3	1	6
---	---	---	---	---	---	---	---

After recursion:  
(not magic 😊)

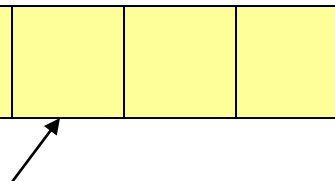
2	4	8	9	1	3	5	6
---	---	---	---	---	---	---	---



Merge:

Use 3 “fingers”  
and 1 more array

1	2	3	4	5			
---	---	---	---	---	--	--	--



(After merge,  
copy back to  
original array)

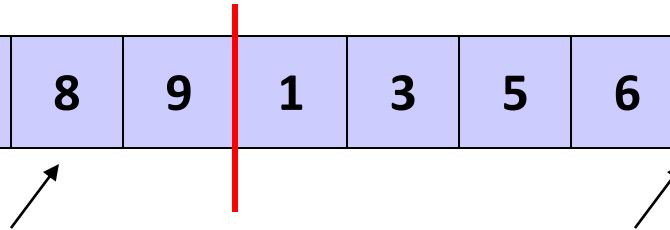
# Merge Sort: Merging Visualization (Soln.)

Start with:

8	2	9	4	5	3	1	6
---	---	---	---	---	---	---	---

After recursion:  
(not magic 😊)

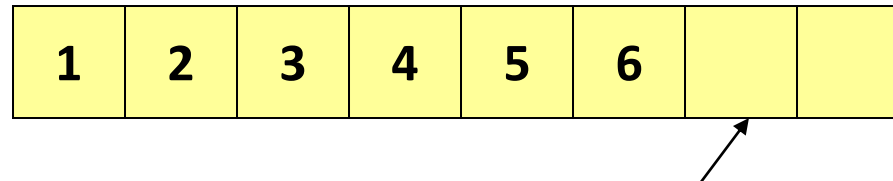
2	4	8	9	1	3	5	6
---	---	---	---	---	---	---	---



Merge:

Use 3 “fingers”  
and 1 more array

1	2	3	4	5	6		
---	---	---	---	---	---	--	--



(After merge,  
copy back to  
original array)

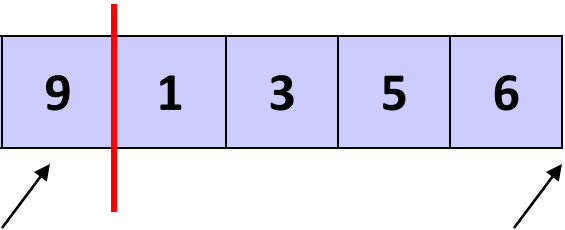
# Merge Sort: Merging Visualization (Soln.)

Start with:

8	2	9	4	5	3	1	6
---	---	---	---	---	---	---	---

After recursion:  
(not magic 😊)

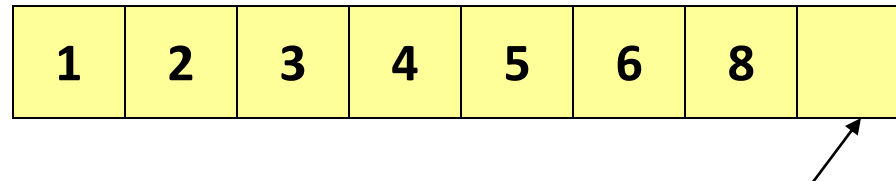
2	4	8	9	1	3	5	6
---	---	---	---	---	---	---	---



Merge:

Use 3 “fingers”  
and 1 more array

1	2	3	4	5	6	8	
---	---	---	---	---	---	---	--



(After merge,  
copy back to  
original array)

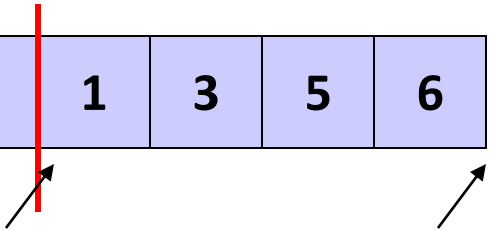
# Merge Sort: Merging Visualization (Soln.)

Start with:

8	2	9	4	5	3	1	6
---	---	---	---	---	---	---	---

After recursion:  
(not magic 😊)

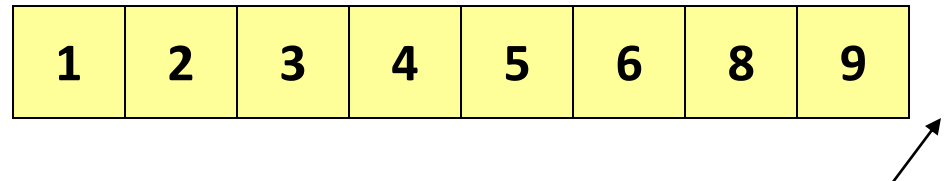
2	4	8	9	1	3	5	6
---	---	---	---	---	---	---	---



Merge:

Use 3 “fingers”  
and 1 more array

1	2	3	4	5	6	8	9
---	---	---	---	---	---	---	---



(After merge,  
copy back to  
original array)

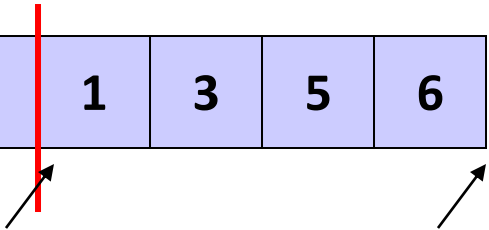
# Merge Sort: Merging Visualization (Soln.)

Start with:

8	2	9	4	5	3	1	6
---	---	---	---	---	---	---	---

After recursion:  
(not magic 😊)


2	4	8	9	1	3	5	6
---	---	---	---	---	---	---	---



Merge:

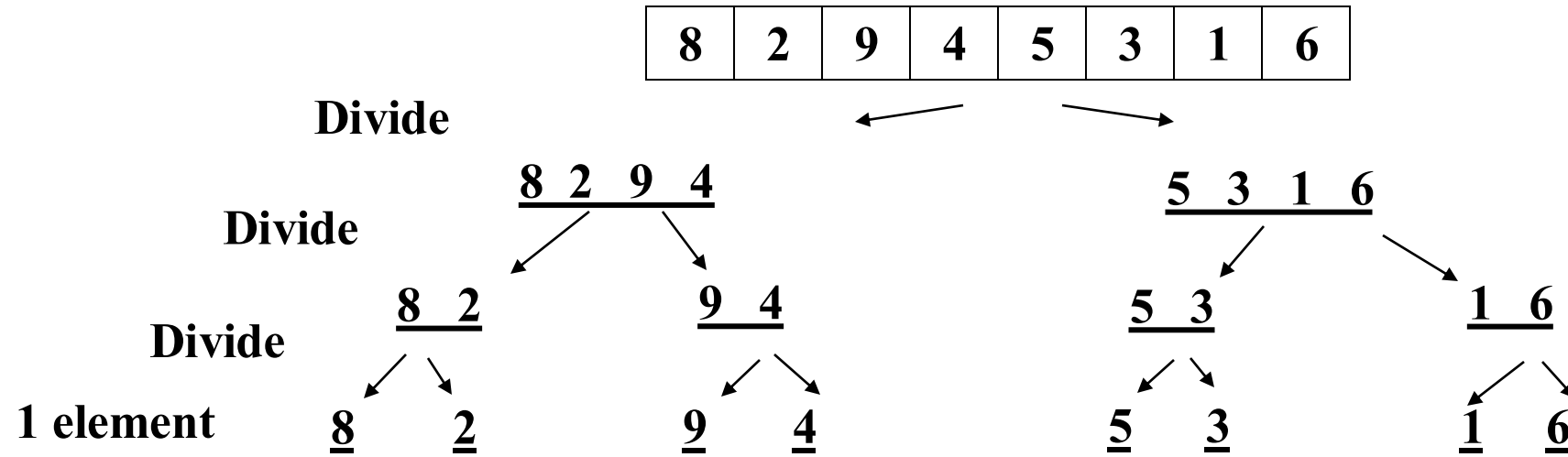
Use 3 “fingers”  
and 1 more array

1	2	3	4	5	6	8	9
---	---	---	---	---	---	---	---

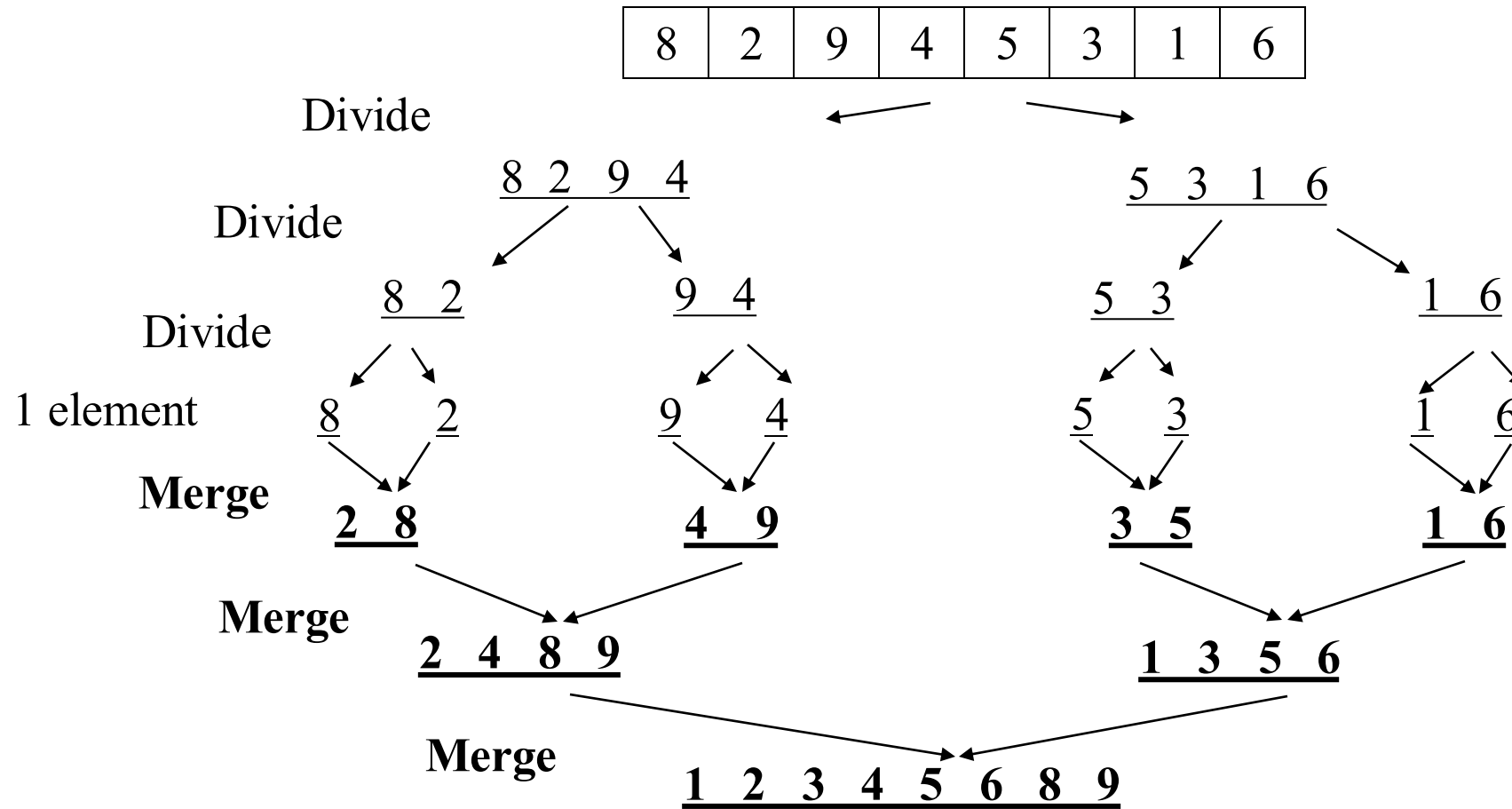


1	2	3	4	5	6	8	9
---	---	---	---	---	---	---	---

# Merge Sort: Splitting Visualization



# Merge Sort: Splitting Visualization

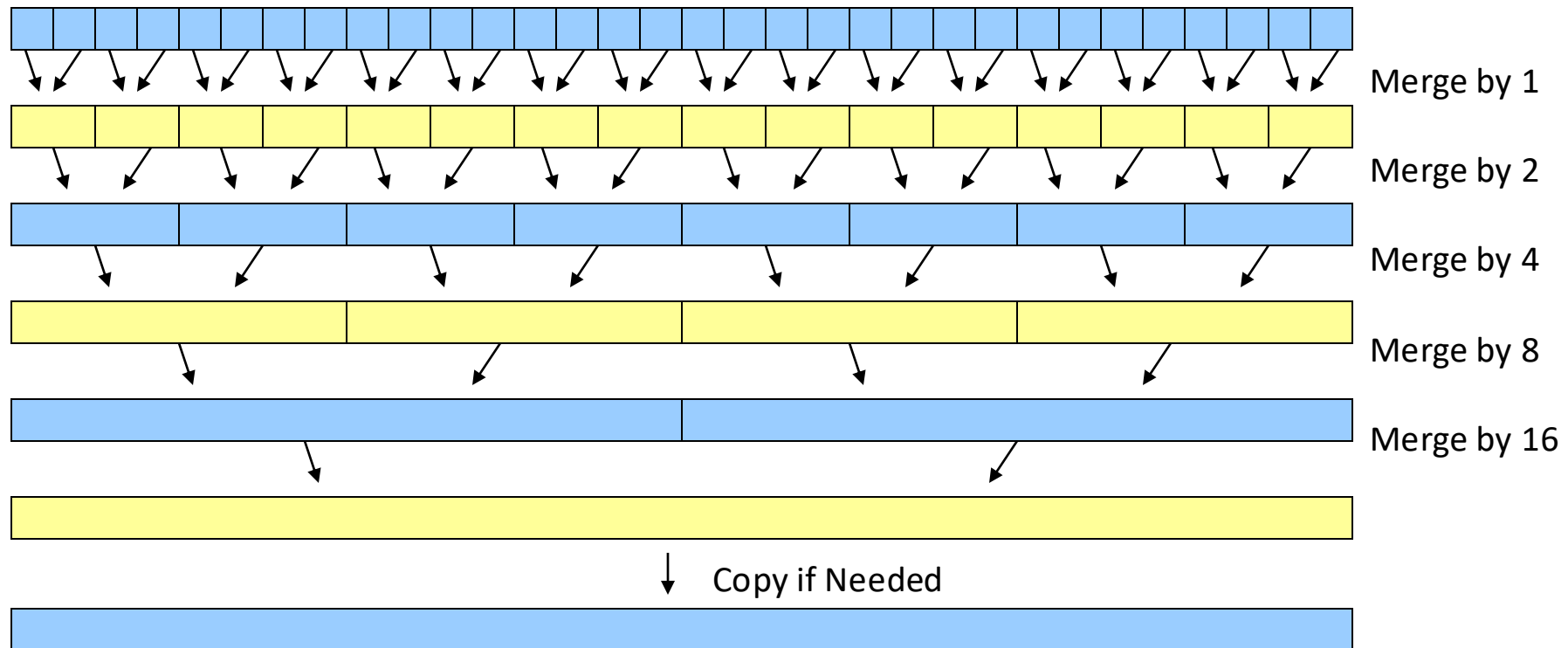


**When a recursive call ends, it's sub-arrays are each in order; just need to merge them in order together**

# Merge Sort: Copy Array Optimization

First recurse down to lists of size 1

As we return from the recursion, switch off arrays





Any Questions?

# Merge Sort: Analysis

## 1. Stable?

- Yes! Just prioritize left array

## 2. In-Place?

- No :(  $\mathcal{O}(n)$  space

## 3. Fast?

- Yes! (in terms of asymptotics)
  - Best Case:  $\mathcal{O}(n \log n)$
  - Worst Case:  $\mathcal{O}(n \log n)$  *Why?*
- **Worse constant factors...**
  - Think: recursive splitting, merging, etc.

# Merge Sort: Runtime Analysis

Recurrence Relation:

$$T(n) = \begin{cases} c_0 & \text{for } n = 1 \\ 2T\left(\frac{n}{2}\right) + c_1n + c_2 & \text{otherwise} \end{cases}$$

Solving:

$$T(n) = 2^{\log n} T(1) + n \log n = n + n \log n \in \mathcal{O}(n \log n)$$

# Today

- Sorting Algorithm 1: Insertion Sort
- Sorting Algorithm 2: Selection Sort
- Sorting Algorithm 3: Heap Sort
  - In-place optimization
- Sorting Algorithm 4: Merge Sort
  - Merging
- Sorting Algorithm 5: Quick Sort
  - Picking a pivot
  - Partioning
- Comparison Sorting Lower Bound

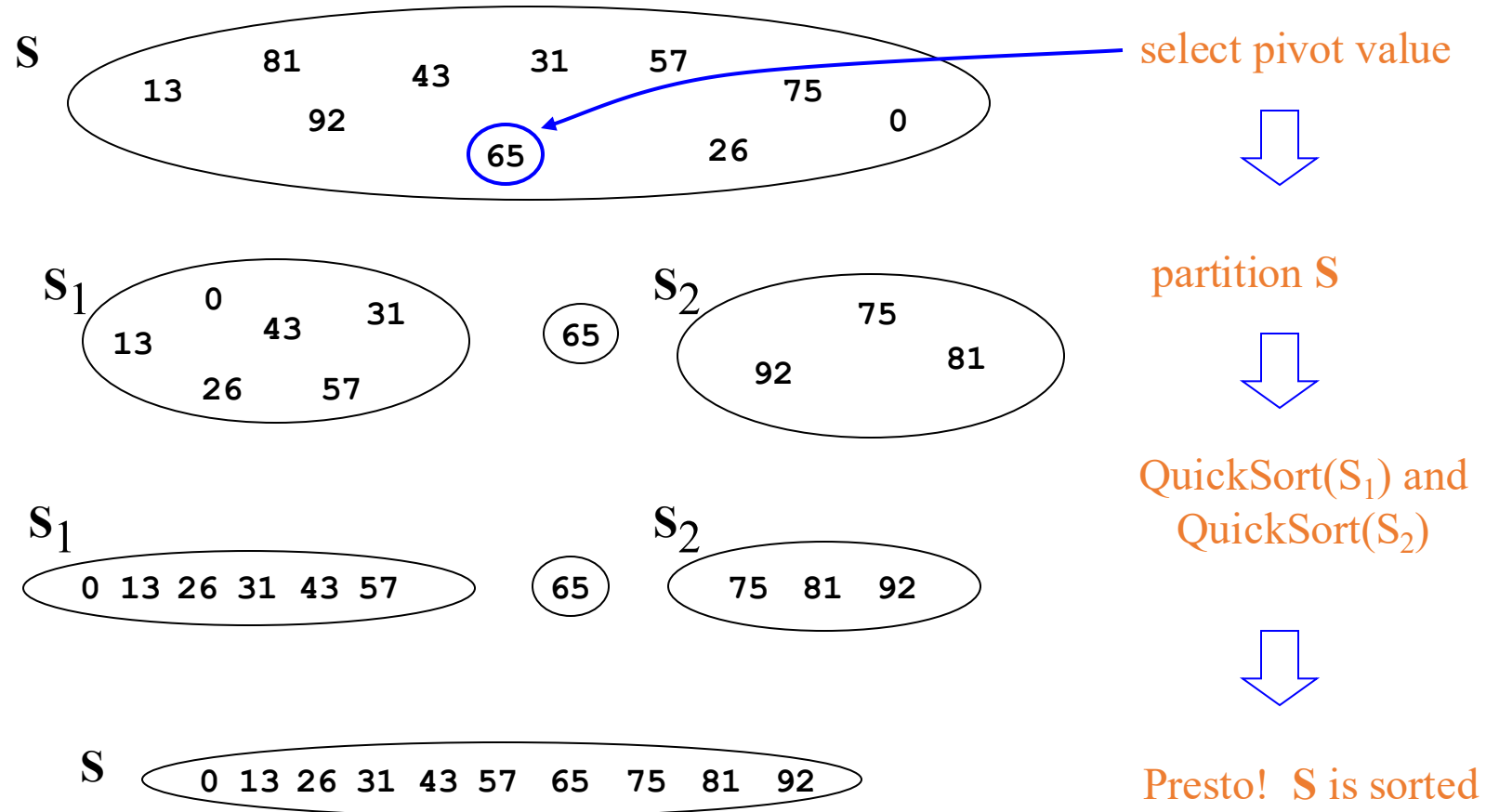
# Quick Sort Warning

There are millions of versions of Quick Sort on the internet. Use ours.

# Sorting Algorithm 5: Quick Sort

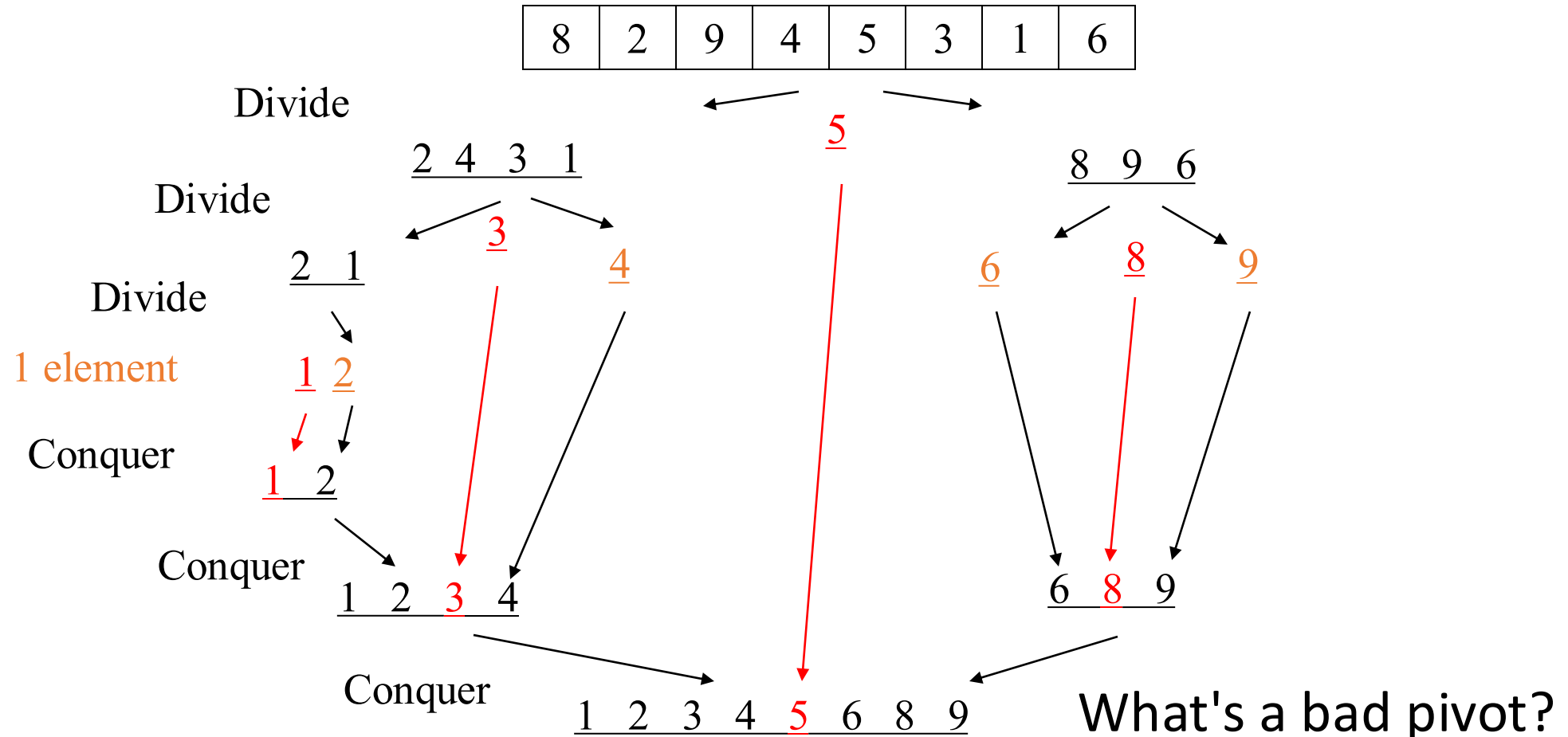
- Algorithm:
  1. Pick a pivot element
    - Hopefully the ~median element
    - Important, performance based on this
  2. Divide elements into 2 "halves":
    - A. less-than pivot
    - B. the pivot
    - C. greater-than pivot
  3. Recursively sort A and C
  4. Sorted output: [**sorted-less-than** then **pivot** then **sorted-greater-than**]

# Quick Sort: Visualization 1



[Weiss]

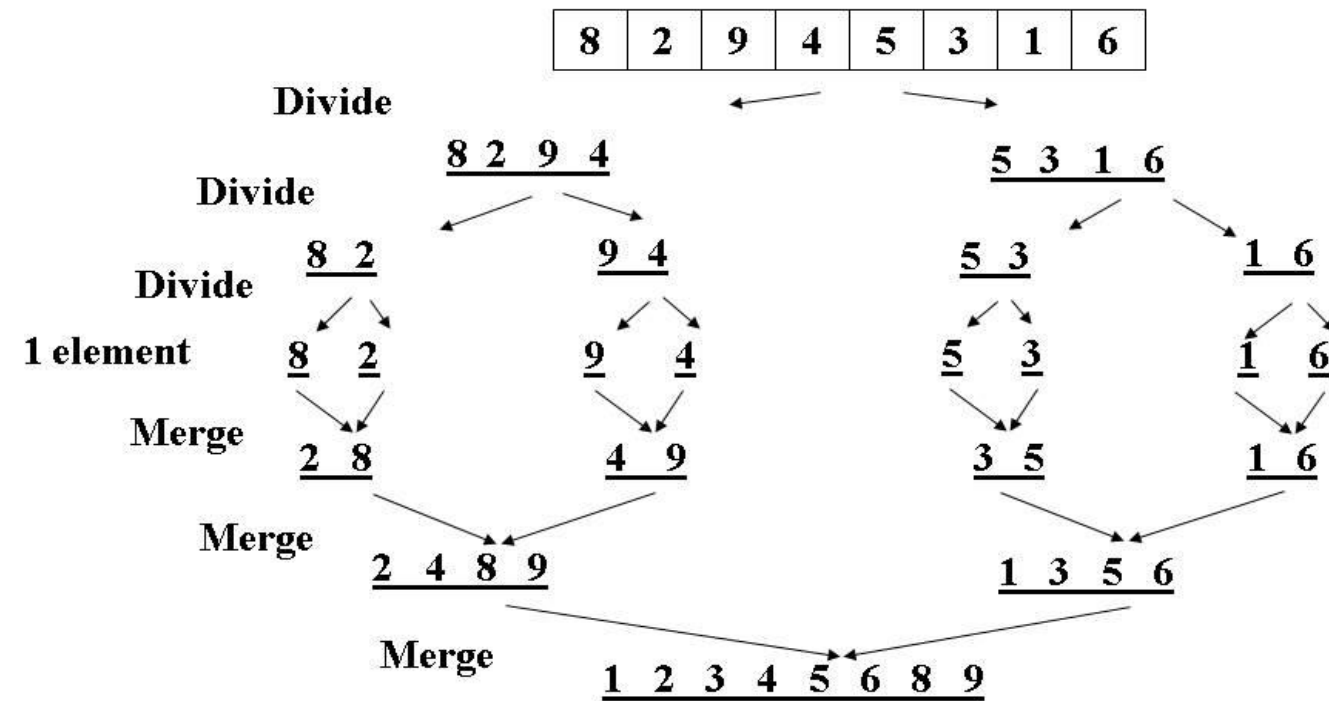
# Quick Sort: Visualization 2



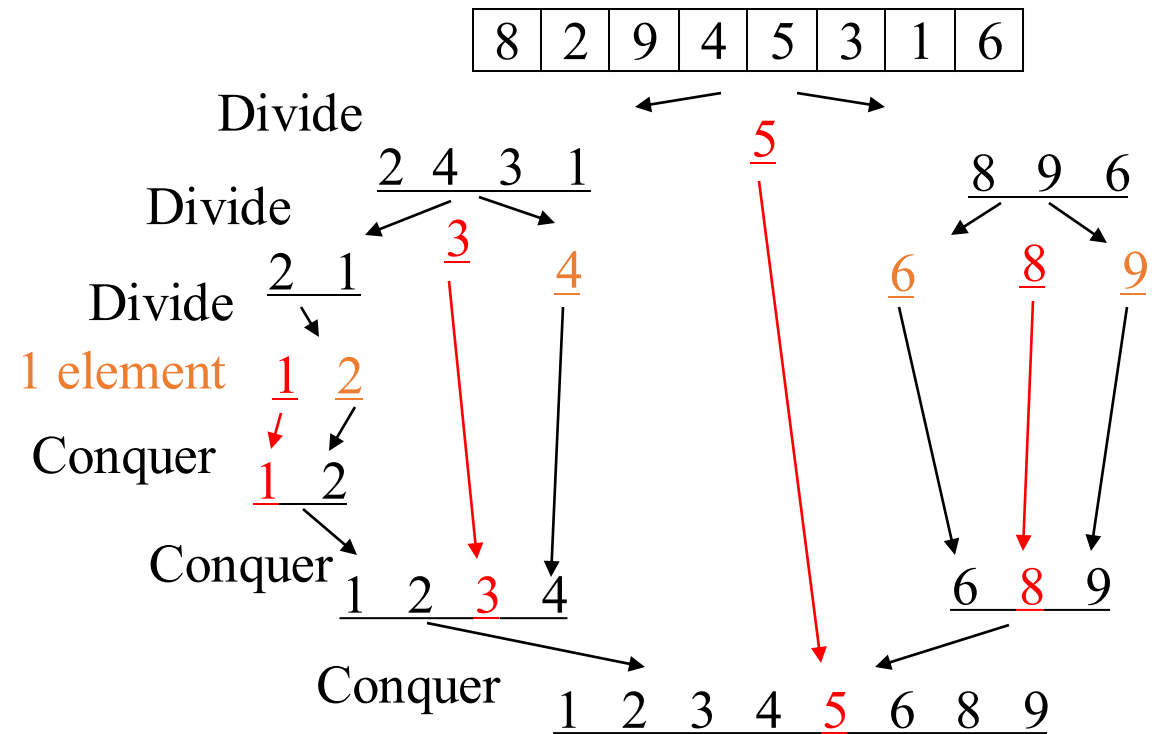


# Merge Sort vs Quick Sort

## MergeSort Recursion Tree

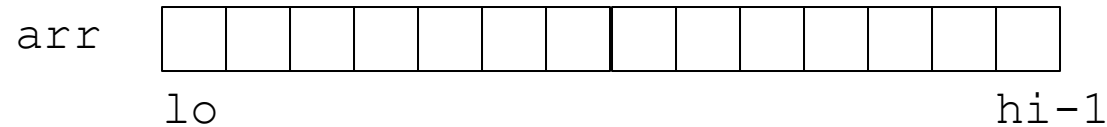


## QuickSort Recursion Tree



# Quick Sort: Picking a (good) Pivot

```
void quicksort(int[] arr, int lo, int hi)
```



1. Option 1: Pick `arr[lo]` or `arr[hi-1]`
  - Fast to pick but likely worst-case (e.g., `arr` is sorted)
2. Option 2: Pick random element
  - Good. But pseudo-randomness is expensive!
3. Option 3: Median of 3
  - e.g., `arr[lo]`, `arr[hi-1]`, `arr[(hi+lo)/2]`
  - Common, tends to work well

# Quick Sort: Partitioning Problem

- Problem: Given good pivot, how to split to two?
  - e.g., [8, 4, 2, 9, 3, 5, 7] and pivot 5,
    - how to split to two - 4, 2, 3 and 8, 9, 7?
- Ideals:
  - Fast  $\mathcal{O}(n)$  linear time
  - In-place

Ideas?

# Quick Sort: "Hoare" Partitioning Approach

1. Swap pivot with `arr[lo]` (i.e., move it out of the way)
2. Use 2 pointers `l` and `r`, starting at `lo+1` and `hi-1`
  - Idea: Move `l` and `r` such that:
    - `arr[l]` should be on the right of pivot and `arr[r]` should be on the left of pivot

```
while (l < r)
    if (arr[l] <= pivot) l++
    else if (arr[r] > pivot) r--
    else swap arr[l] and arr[r]
```
3. Put pivot back in middle (Swap with `arr[r]`)

# Quick Sort: Example

Pick pivot 7, median of 3

3	4	9	1	7	0	5	2	6	8
---	---	---	---	---	---	---	---	---	---

Start "Hoare" Partition:

3	4	9	1	7	0	5	2	6	8
---	---	---	---	---	---	---	---	---	---

Move 7, init l and r:

7	4	9	1	3	0	5	2	6	8
---	---	---	---	---	---	---	---	---	---

l

r

Move l and r:

7	4	9	1	3	0	5	2	6	8
---	---	---	---	---	---	---	---	---	---

l

r

Swap `arr[l]` and `arr[r]`:

7	4	6	1	3	0	5	2	9	8
---	---	---	---	---	---	---	---	---	---

l

r

# Quick Sort: Example (cont.)

After swap:

7	4	6	1	3	0	5	2	9	8
---	---	---	---	---	---	---	---	---	---

l

r

Move l and r:

7	4	6	1	3	0	5	2	9	8
---	---	---	---	---	---	---	---	---	---

r

l

$r \leq l$ , move pivot back

7	4	6	1	3	0	5	2	9	8
---	---	---	---	---	---	---	---	---	---

r

l

"Hoare" Partitioned!

2	4	6	1	3	0	5	7	9	8
---	---	---	---	---	---	---	---	---	---

Any Questions?

# Quick Sort: Analysis

## 1. Stable?

- No :(

## 2. In-Place?

- Yes!

## 3. Fast?

- Yes! (in terms of asymptotics)
  - Best Case:  $\mathcal{O}(n \log n)$
  - Average Case:  $\mathcal{O}(n \log n)$  (when good pivot)
  - Worst Case:  $\mathcal{O}(n^2)$  *Why?*
- **Worse constant factors...**
  - Think: recursive splitting, merging, etc.
- In practice: way, way better



# Quick Sort: Runtime Analysis

Best Case:

$$T(n) = \begin{cases} c_0 & \text{for } n = 0 \text{ or } 1 \\ 2T\left(\frac{n}{2}\right) + c_1n + c_2 & \text{otherwise} \end{cases}$$

Worst Case:

$$T(n) = \begin{cases} c_0 & \text{for } n = 0 \text{ or } 1 \\ T(n-1) + c_1n + c_2 & \text{otherwise} \end{cases}$$

Average Case (good pivot):

$$T(n) \in \mathcal{O}(n \log n)$$

Proof is in the textbook, Weiss 7.7

# Comparison Sorting: CUTOFF Strategy

```
void sort(int[] arr, int lo, int hi) {  
    if (hi - lo < CUTOFF)  
        insertionSort(arr, lo, hi); // or Selection Sort  
    else  
        quickSort(arr, lo, hi) // or Merge Sort, etc.  
}
```

# Comparison Sorting: Comparisons

	Run-time	Stable?	In-Place?
<b>Insertion Sort</b>	Best Case: $\mathcal{O}(n)$ Worst Case: $\mathcal{O}(n^2)$ Average Case: $\mathcal{O}(n^2)$	Stable	In-place
<b>Selection Sort</b>	$\mathcal{O}(n^2)$	Not Stable	In-place
<b>Heap Sort</b>	$\mathcal{O}(n \log n)$	Not Stable	In-place
<b>Merge Sort</b>	$\mathcal{O}(n \log n)$	Stable	Not In-place
<b>Quick Sort</b> ("Hoare" Partition)	Best Case: $\mathcal{O}(n \log n)$ Worst Case: $\mathcal{O}(n^2)$ Average Case: $\mathcal{O}(n \log n)$	Not Stable	In-place

# Today

- Sorting Algorithm 1: Insertion Sort
- Sorting Algorithm 2: Selection Sort
- Sorting Algorithm 3: Heap Sort
  - In-place optimization
- Sorting Algorithm 4: Merge Sort
  - Merging
- Sorting Algorithm 5: Quick Sort
  - Picking a pivot
  - Partitioning
- Comparison Sorting Lower Bound

# Comparison Sorting Lower Bound

We keep hitting  $\mathcal{O}(n \log n)$  in the worst case.

Can we do better?

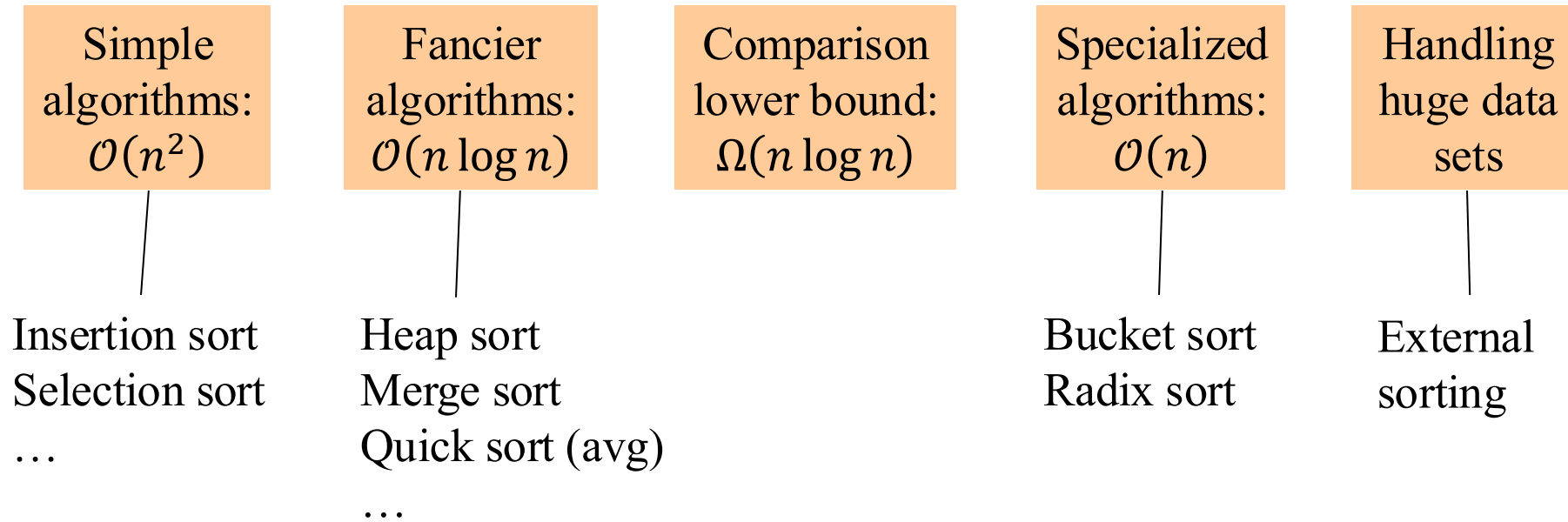
Or is this  $\mathcal{O}(n \log n)$  pattern a fundamental barrier?

Without more information about our data set, we cannot do better.

- i.e. assume all we know about the input type is that it has a `compareTo()` method.

<b>Comparison Sorting Lower Bound</b>
Any sorting algorithm that knows nothing about the input data type, except how to compare two instances, must take $\Omega(n \log n)$ time in the worst case.

# Sorting: The Big Picture



Any Questions?