

# Sorting

Ye Yang

Stevens Institute of Technology

# So far...

- We have covered stacks, queues, priority queues, and hash tables
  - Emphasis on providing one element at a time
- We will now step away from ADTs and talk about *sorting algorithms*
- Note that we have already implicitly met sorting
  - Priority Queues
  - Binary Search and Binary Search Trees
- Sorting benefitted and limited ADT performance
  - No single “best” sort for all scenarios
  - Knowing “one way to sort” is not sufficient

# A Comparison Sort Algorithm

We have  $n$  comparable elements in an array, and we want to rearrange them to be in *increasing order*

Input:

- An array  $A$  of data records
- A key value in each data record (maybe many fields)
- A comparison function (must be consistent and total): Given keys  $a$  and  $b$  is  $a < b$ ,  $a = b$ ,  $a > b$ ?

Effect:

- Reorganize the elements of  $A$  such that for any  $i$  and  $j$  such that if  $i < j$  then  $A[i] \leq A[j]$
- Array  $A$  must have all the data it started with

# Arrays? Just Arrays?

The algorithms we will talk about will assume that the data is an array

- Arrays allow direct index referencing
- Arrays are contiguous in memory

But data may come in a linked list

- Some algorithms can be adjusted to work with linked lists but algorithm performance will likely change (at least in constant factors)
- May be reasonable to do a  $O(n)$  copy to an array and then back to a linked list

# Further Concepts / Extensions

## Stable sorting:

- Duplicate data is possible
- Does not change duplicate's original ordering relative to each other

## In-place sorting:

- Keeps the sorted items in the same array that initially held the input items
- Uses at most  $O(1)$  auxiliary space beyond initial array

## Non-Comparison Sorting:

- Redefining the concept of comparison to improve speed

## Other concepts:

- External Sorting: Too much data to fit in main memory
- Parallel Sorting: When you have multiple processors

# **STANDARD COMPARISON SORT ALGORITHMS**

# Sorting: The Big Picture

Horrible  
algorithms:  
 $\Omega(n^2)$

Bogo Sort  
Stooge Sort

Simple  
algorithms:  
 $O(n^2)$

Insertion sort  
Selection sort  
Bubble Sort

Shell sort

...

Fancier  
algorithms:  
 $O(n \log n)$

Heap sort  
Merge sort  
Quick sort (avg)

...

Comparison  
lower bound:  
 $\Omega(n \log n)$

Specialized  
algorithms:  
 $O(n)$

Bucket sort  
Radix sort

# Sorting: The Big Picture

Horrible  
algorithms:  
 $\Omega(n^2)$

Bogo Sort  
Stooge Sort



*Read about on your own to learn how not to sort data*

Simple  
algorithms:  
 $O(n^2)$

Insertion sort  
Selection sort  
Bubble Sort

Shell sort

...

Fancier  
algorithms:  
 $O(n \log n)$

Heap sort  
Merge sort  
Quick sort (avg)

...

Comparison  
lower bound:  
 $\Omega(n \log n)$

Specialized  
algorithms:  
 $O(n)$

Bucket sort  
Radix sort



# Sorting: The Big Picture

Horrible  
algorithms:  
 $\Omega(n^2)$

Bogo Sort  
Stooge Sort

Simple  
algorithms:  
 $O(n^2)$

Insertion sort  
Selection sort  
Bubble Sort

Shell sort

...

Fancier  
algorithms:  
 $O(n \log n)$

Heap sort  
Merge sort  
Quick sort (avg)

...

Comparison  
lower bound:  
 $\Omega(n \log n)$

Specialized  
algorithms:  
 $O(n)$

Bucket sort  
Radix sort

# Insertion Sort

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

Alternate way of saying this:

- Sort first element (this is easy)
- Now insert 2<sup>nd</sup> element in order
- Now insert 3<sup>rd</sup> element in order
- Now insert 4<sup>th</sup> element in order
- ...

Loop invariant:

When loop index is  $i$ , first  $i$  elements are sorted

Time? Best: \_\_\_\_\_ Worst: \_\_\_\_\_ Average: \_\_\_\_\_

# Insertion Sort

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

Alternate way of saying this:

- Sort first element (this is easy)
- Now insert 2<sup>nd</sup> element in order
- Now insert 3<sup>rd</sup> element in order
- Now insert 4<sup>th</sup> element in order
- ...

Loop invariant:

When loop index is  $i$ , first  $i$  elements are sorted  
*Already or Nearly*

	<i>Sorted</i>	<i>Reverse Sorted</i>	<i>See Book</i>
Time:	Best: $O(n)$	Worst: $O(n^2)$	Average: $O(n^2)$
	Stable and In-Place		

# Implementing Insertion Sort

There's a trick to doing the insertions without crazy array re-shifting

```
void insertionSort(int[] arr) {  
    for(int i = 1; i < arr.length; i++) {  
        int tmp = arr[i];  
        int j;  
        for( j = i; j > 0 && tmp < arr[j-1]; j-- )  
            arr[j] = arr[j-1];  
        arr[j] = tmp;  
    }  
}
```

As with heaps, “moving the hole” is faster than unnecessary swapping (impacts constant factor)

# Selection Sort

Idea: At step  $k$ , find the **smallest** element among the unsorted elements and put it at position  $k$

Alternate way of saying this:

- Find smallest element, put it 1st
- Find next smallest element, put it 2nd
- Find next smallest element, put it 3rd
- ...

Loop invariant:

When loop index is  $i$ , the first  $i$  elements are the  $i$  smallest elements in sorted order

Time? Best: \_\_\_\_\_ Worst: \_\_\_\_\_ Average: \_\_\_\_\_

# Selection Sort

Idea: At step  $k$ , find the smallest element among the unsorted elements and put it at position  $k$

Alternate way of saying this:

- Find smallest element, put it 1st
- Find next smallest element, put it 2nd
- Find next smallest element, put it 3rd
- ...

Loop invariant:

When loop index is  $i$ , the first  $i$  elements are the  $i$  smallest elements in sorted order

Time: Best:  $O(n^2)$  Worst:  $O(n^2)$  Average:  $O(n^2)$

Recurrence Relation:  $T(n) = n + T(N-1)$ ,  $T(1) = 1$

Stable and In-Place

# Implementing SelectionSort

```
void selectionSort(int[] arr) {  
    for(int i = 0; i < arr.length-1; i++) {  
        int min_idx = i;  
        for( int j = i+1; j < arr.length; j++ )  
            if (arr[j] < arr[min_idx])  
                min_idx = j;  
  
        int temp = arr[min_idx];  
        arr[min_idx] = arr[i];  
        arr[i] = temp;  
    }  
}
```

If input is a data structure faster than an array, we call it...

# Bubble Sort

- Idea:
  - Compares adjacent array elements and exchanges their values if they are out of order
  - Smaller values bubble up to the top of the array and larger values sink to the bottom; hence the name



# Implementation of Bubble Sort

```
void bubbleSort(int[] arr) {  
    int n = arr.length;  
    for (int i = 0; i < n-1; i++)  
        for (int j = 0; j < n-i-1; j++)  
            if (arr[j] > arr[j+1])  
            {  
                // swap temp and arr[i]  
                int temp = arr[j];  
                arr[j] = arr[j+1];  
                arr[j+1] = temp;  
            }  
}
```

# Sorting: The Big Picture

Horrible  
algorithms:  
 $\Omega(n^2)$

Bogo Sort  
Stooge Sort

Simple  
algorithms:  
 $O(n^2)$

Insertion sort  
Selection sort  
Bubble Sort

Shell sort

...

Fancier  
algorithms:  
 $O(n \log n)$

Heap sort  
Merge sort  
Quick sort (avg)

...

Comparison  
lower bound:  
 $\Omega(n \log n)$

Specialized  
algorithms:  
 $O(n)$

Bucket sort  
Radix sort

# Heap Sort

Sorting with a heap is easy:

```
buildHeap(...);  
for(i=0; i < arr.length; i++)  
    arr[i] = deleteMin();
```

Worst-case running time:  $O(n \log n)$  Why?

We have the array-to-sort and the heap

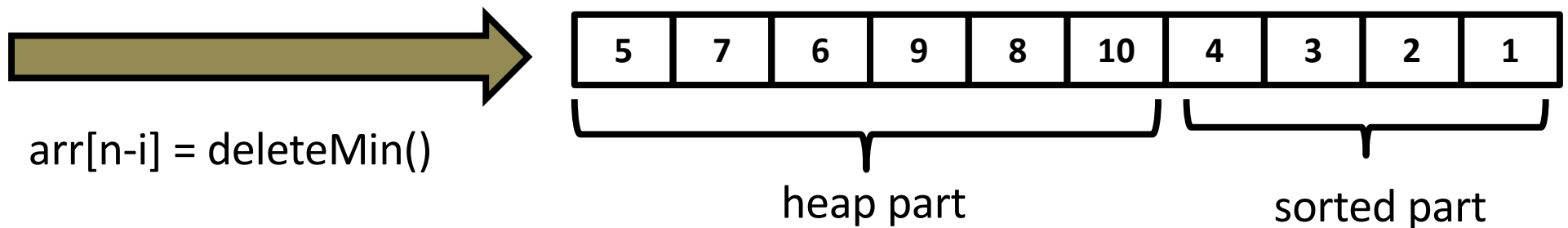
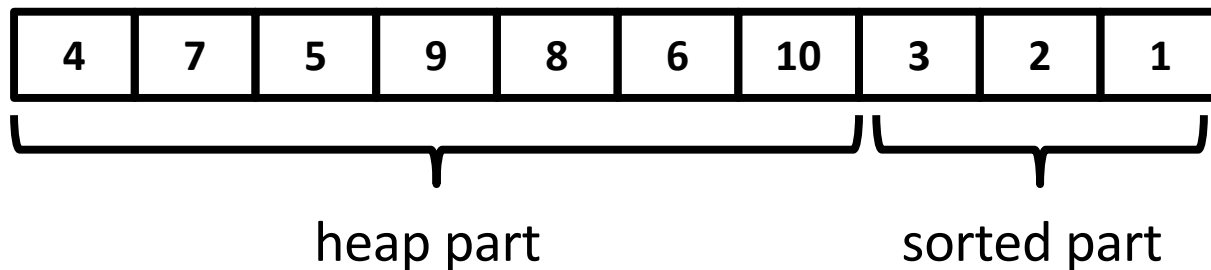
- So this is neither an in-place or stable sort
- There's a trick to make it in-place

# In-Place Heap Sort

Treat initial array as a heap (via buildHeap)

When you delete the  $i^{\text{th}}$  element,

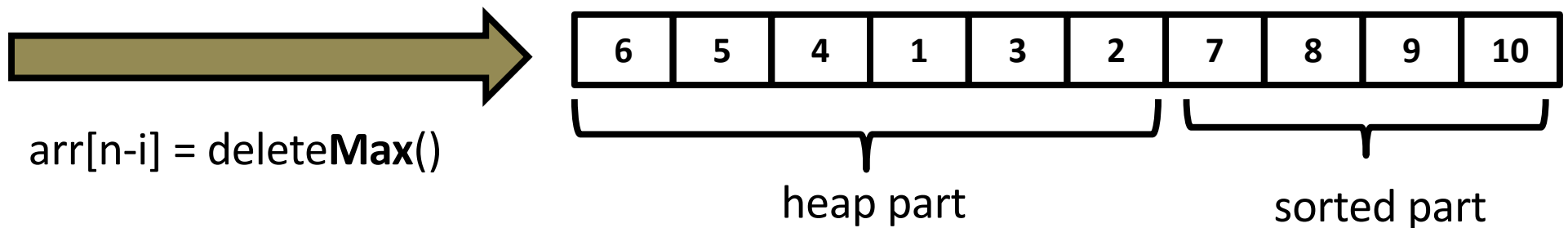
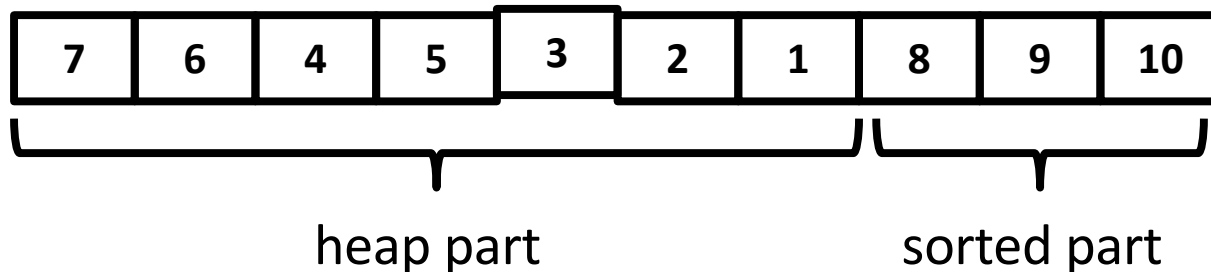
Put it at  $\text{arr}[n-i]$  since that array location is not part of the heap anymore!



# In-Place Heap Sort

But this reverse sorts... how to fix?

Build a maxHeap instead: e.g. [10, 6, 9, 5, 3, 8, 7, 4, 6]. After three deleteMax:



# Divide and Conquer

Very important technique in algorithm design

1. Divide problem into smaller parts
2. Independently solve the simpler parts
  - Think recursion
  - Or potential parallelism
3. Combine solution of parts to produce overall solution

# Divide-and-Conquer Sorting

# Two great sorting methods are fundamentally divide-and-conquer

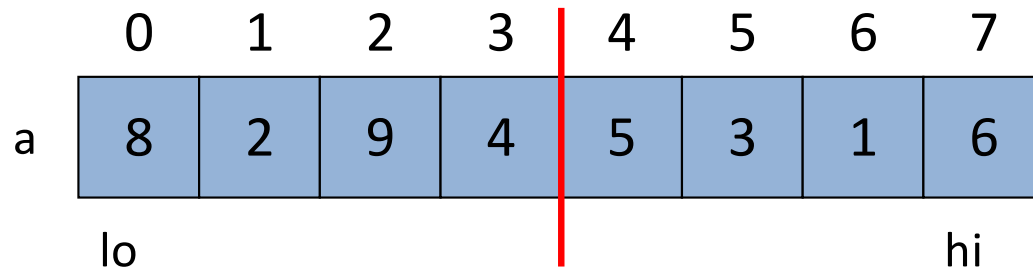
Mergesort:

- Recursively sort the left half
- Recursively sort the right half
- Merge the two sorted halves

Quicksort:

- Pick a “pivot” element
- Separate elements by pivot ( $<$  and  $>$ )
- Recursive on the separations
- Return  $< \text{pivot}, \text{pivot}, > \text{pivot}$ ]

# Mergesort



To sort array from position **lo** to position **hi**:

- If range is 1 element long, it is already sorted! (our base case)
- Else, split into two halves:
  - Sort from **lo** to **(hi+lo)/2**
  - Sort from **(hi+lo)/2** to **hi**
  - Merge the two halves together

Merging takes two sorted parts and sorts everything

- $O(n)$  but requires auxiliary space...



# Example: Focus on Merging

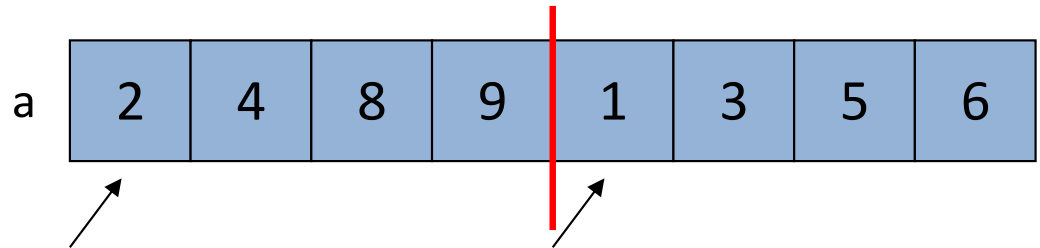
Start with:

After recursion:

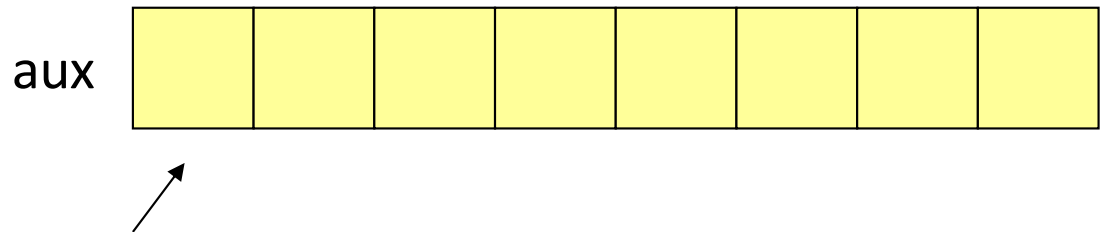


Merge:

Use 3 “fingers” and 1  
more array



After merge, we will  
copy back to the  
original array



# Example: Focus on Merging

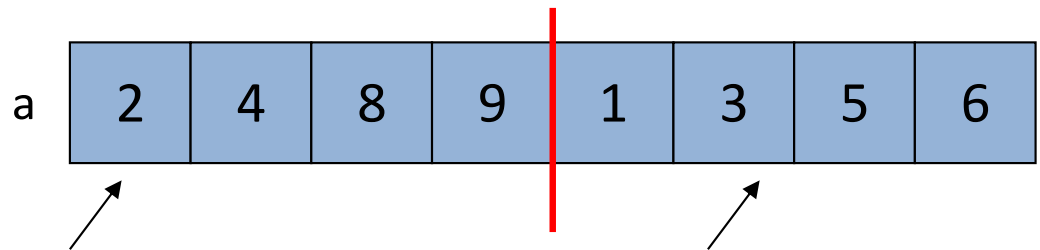
Start with:

After recursion:

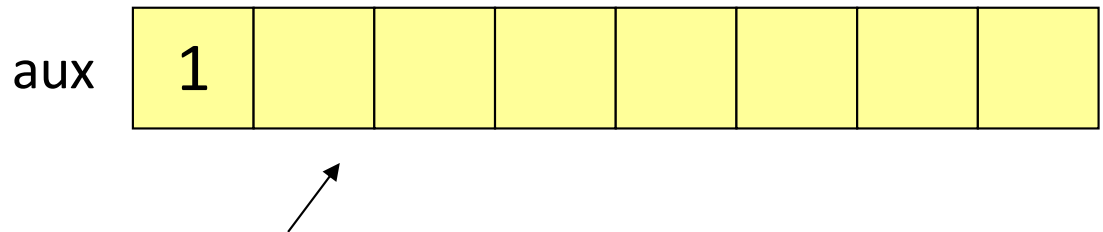


Merge:

Use 3 “fingers” and 1  
more array



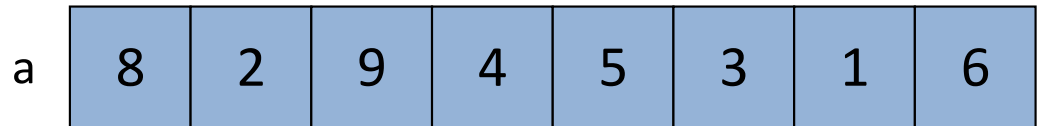
After merge, we will  
copy back to the  
original array



# Example: Focus on Merging

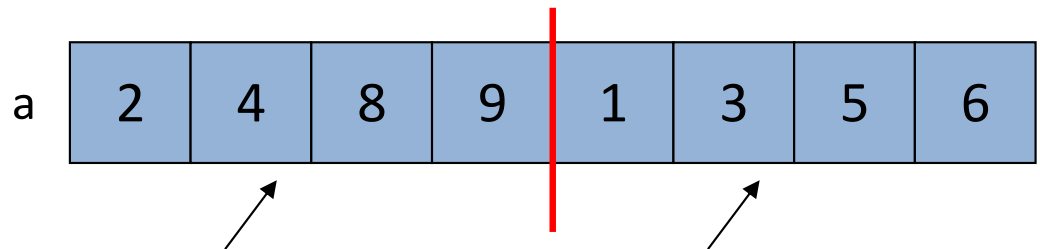
Start with:

After recursion:

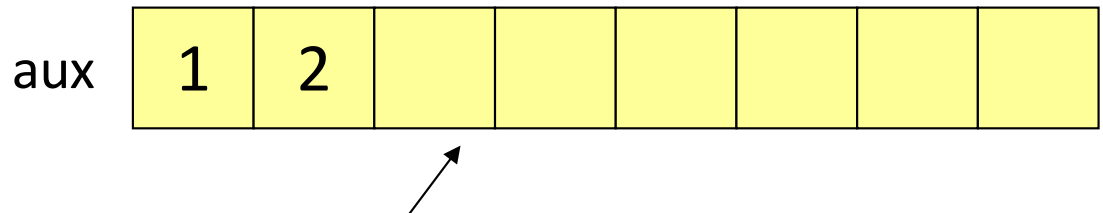


Merge:

Use 3 “fingers” and 1 more array



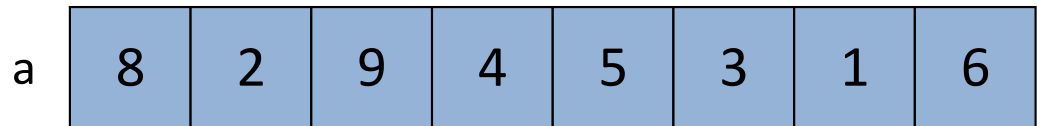
After merge, we will copy back to the original array



# Example: Focus on Merging

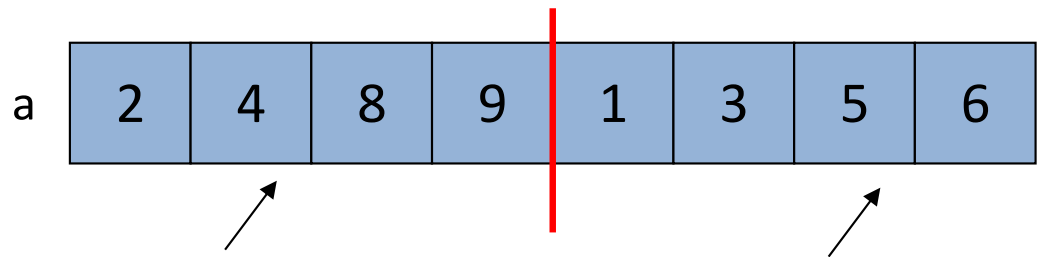
Start with:

After recursion:

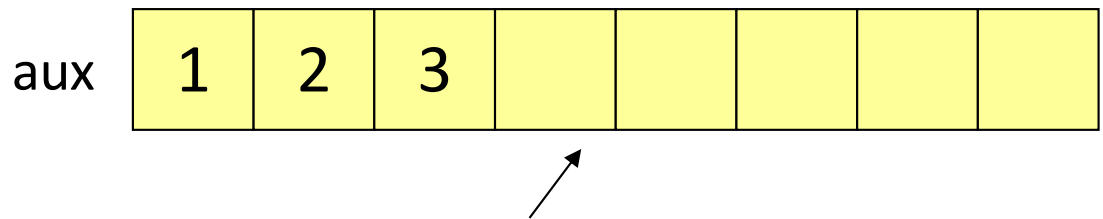


Merge:

Use 3 “fingers” and 1  
more array



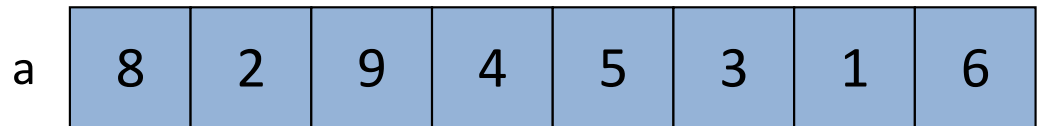
After merge, we will  
copy back to the  
original array



# Example: Focus on Merging

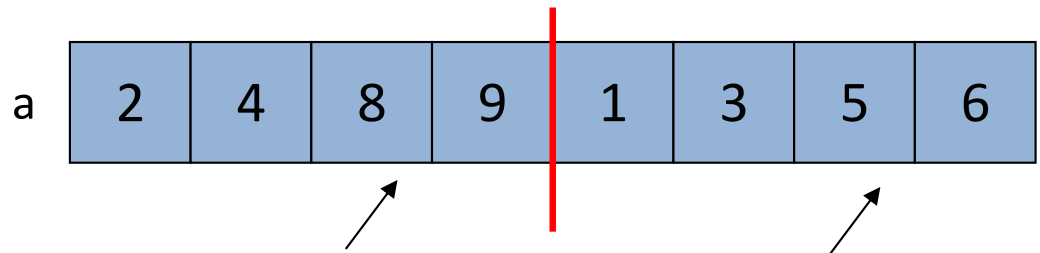
Start with:

After recursion:

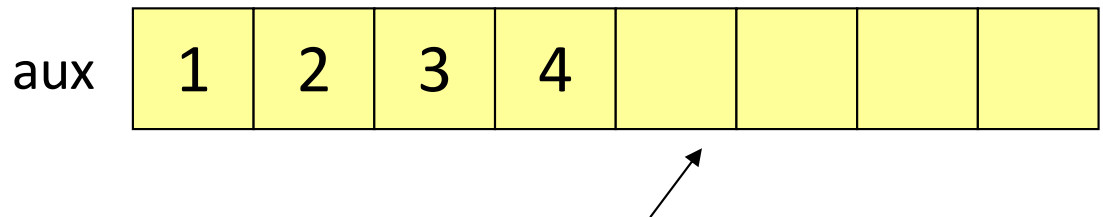


Merge:

Use 3 “fingers” and 1  
more array



After merge, we will  
copy back to the  
original array



# Example: Focus on Merging

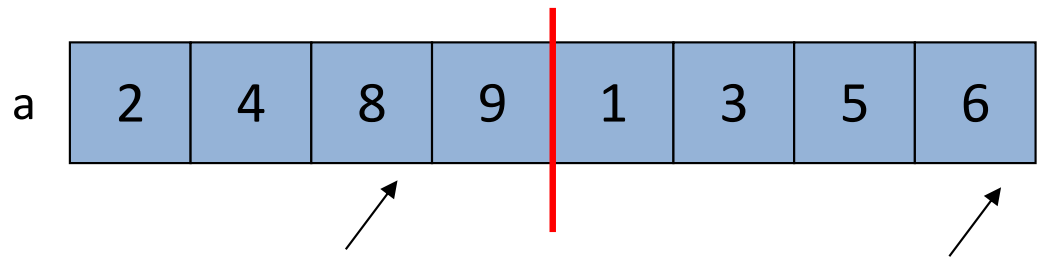
Start with:

After recursion:

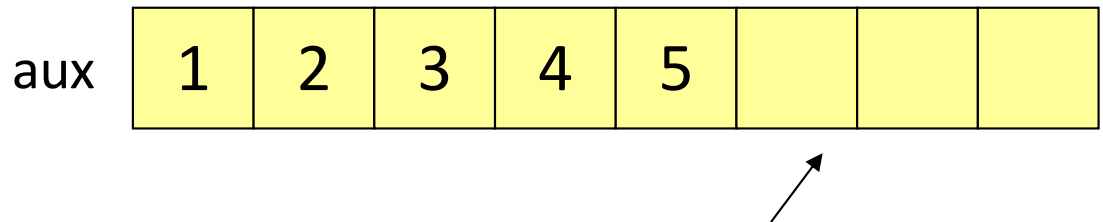


Merge:

Use 3 “fingers” and 1  
more array



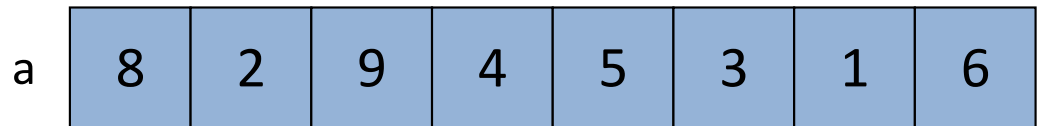
After merge, we will  
copy back to the  
original array



# Example: Focus on Merging

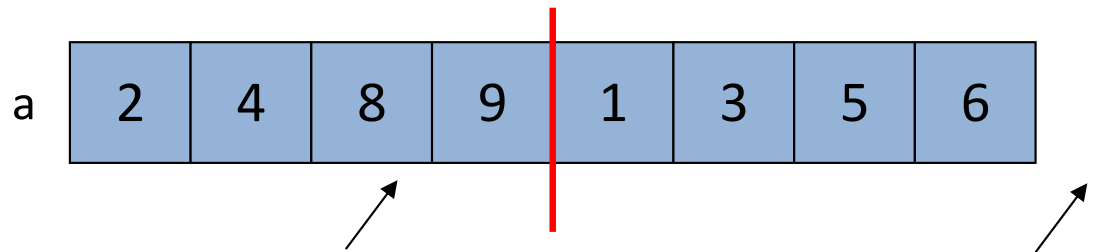
Start with:

After recursion:

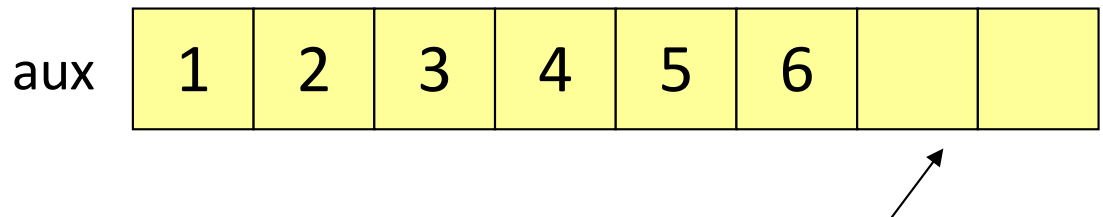


Merge:

Use 3 “fingers” and 1 more array



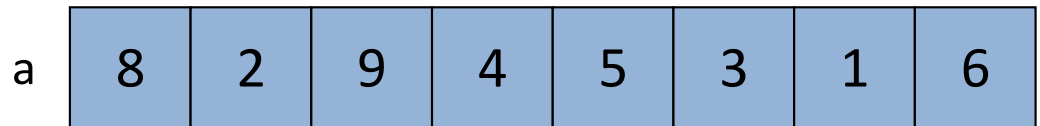
After merge, we will copy back to the original array



# Example: Focus on Merging

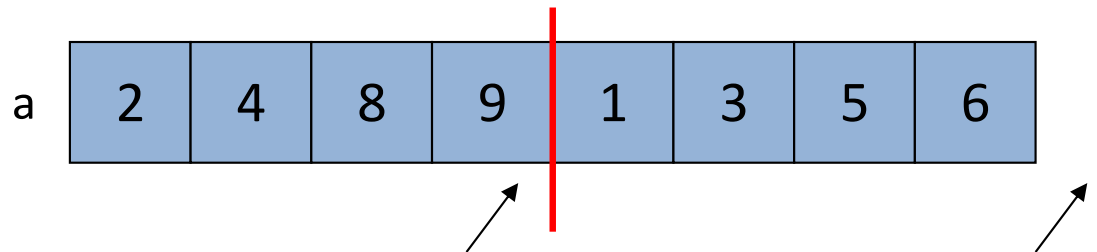
Start with:

After recursion:

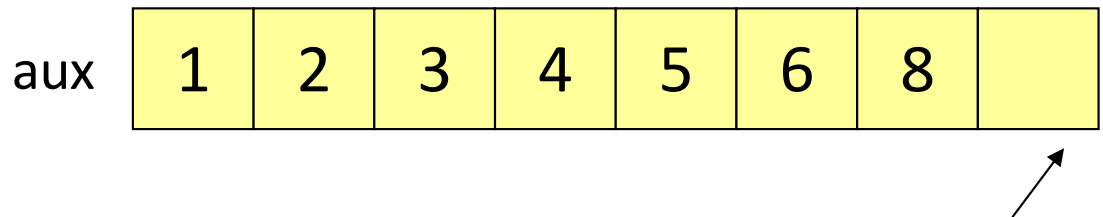


Merge:

Use 3 “fingers” and 1  
more array



After merge, we will  
copy back to the  
original array

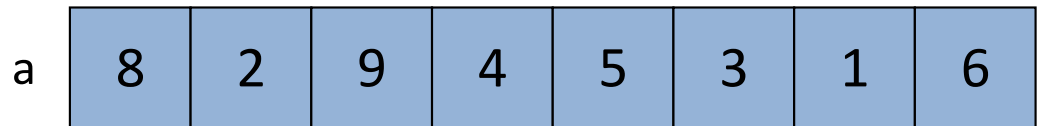




# Example: Focus on Merging

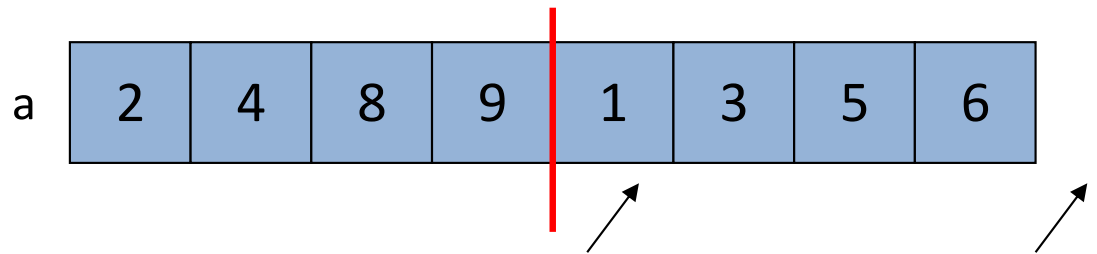
Start with:

After recursion:

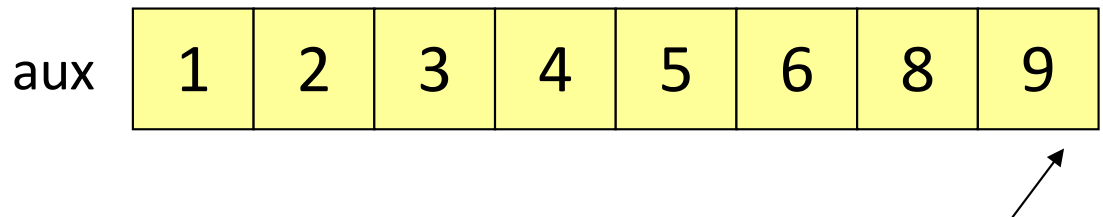


Merge:

Use 3 “fingers” and 1  
more array

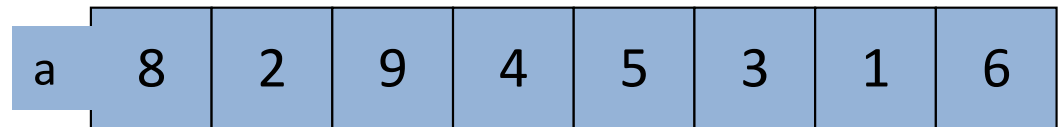


After merge, we will  
copy back to the  
original array

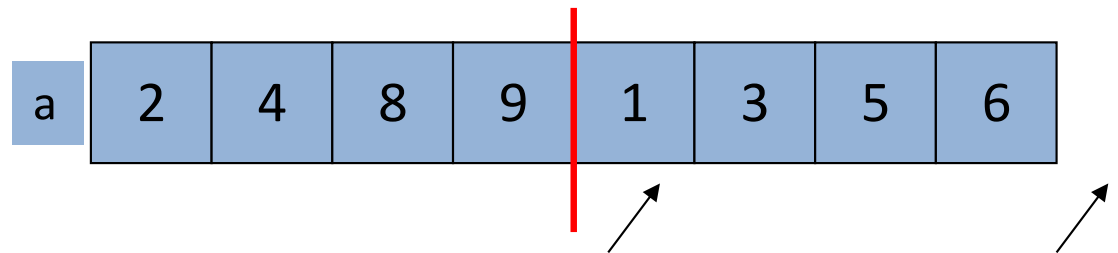


# Example: Focus on Merging

Start with:

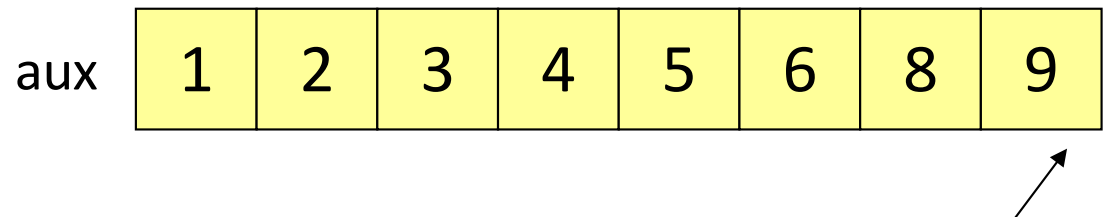


After recursion:

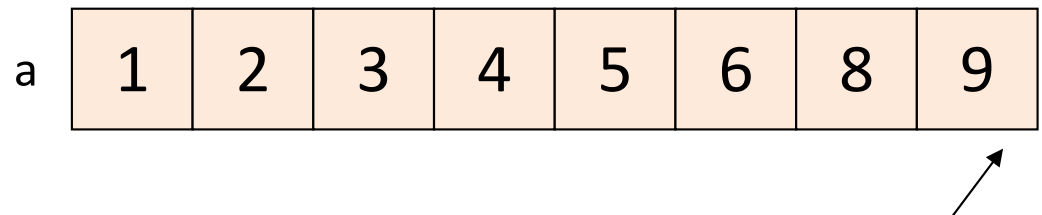


Merge:

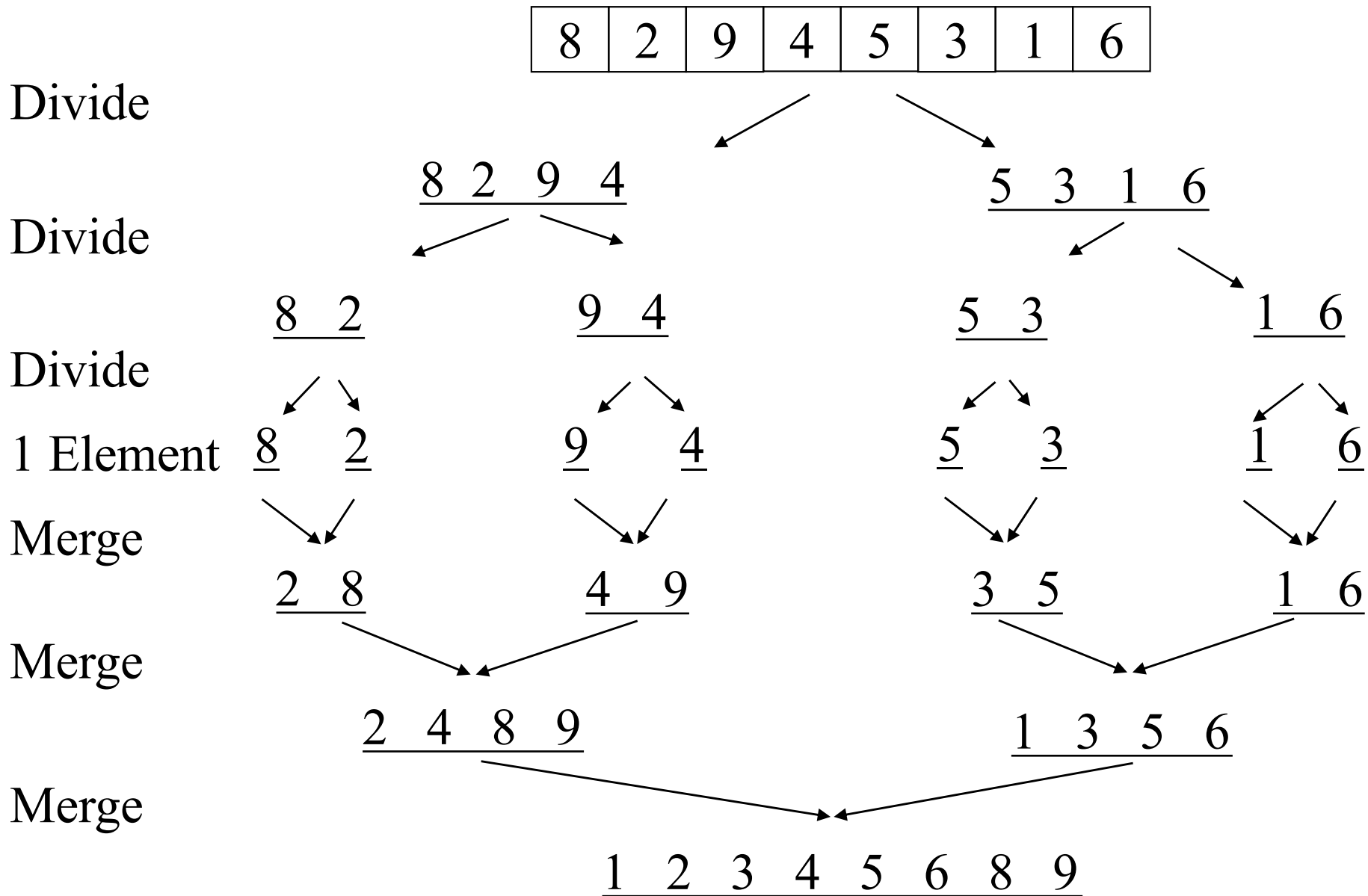
Use 3 “fingers” and 1 more array



After merge, we will copy back to the original array

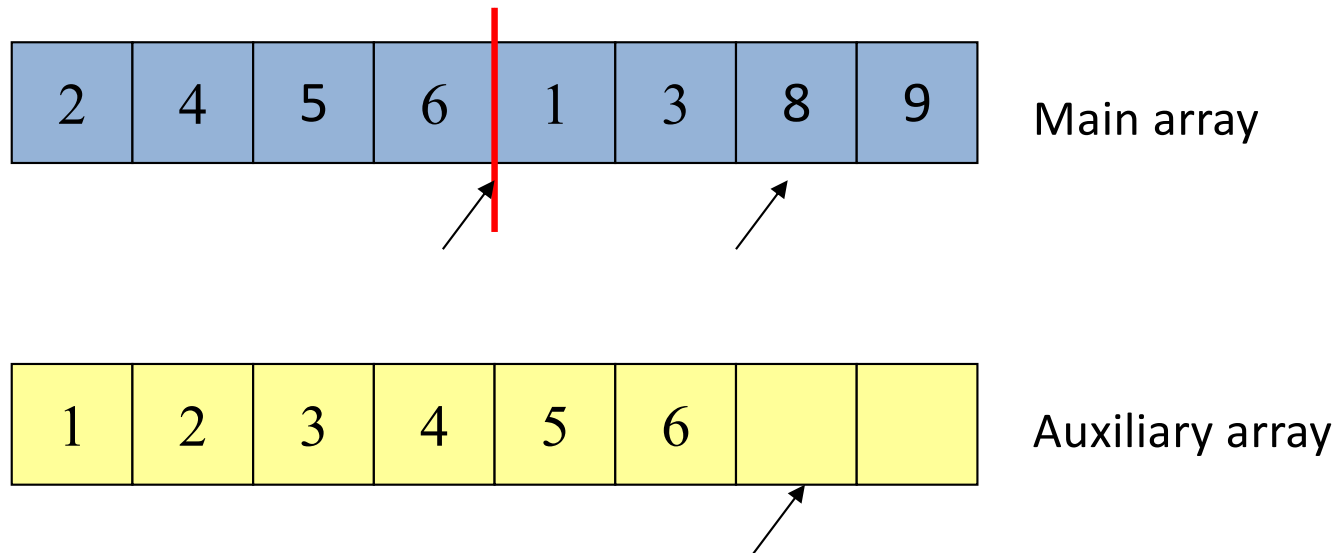


# Example: Mergesort Recursion



# Mergesort: Time Saving Details

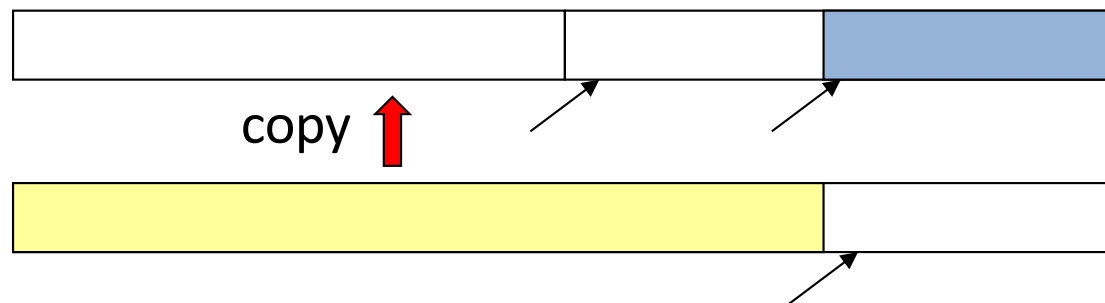
What if the final steps of our merge looked like this?



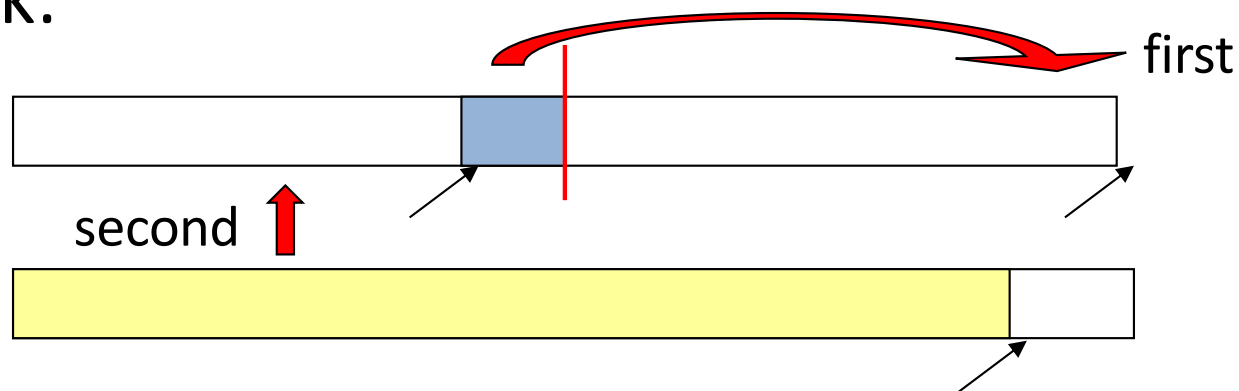
Isn't it wasteful to copy to the auxiliary array just to copy back...

# Mergesort: Time Saving Details

If left-side finishes first, just stop the merge and copy back:



If right-side finishes first, copy drags into right then copy back:



# Mergesort: Space Saving Details

## Simplest / Worst Implementation:

- Use a new auxiliary array of size (hi-lo) for every merge

## Better Implementation

- Use a new auxiliary array of size  $n$  for every merge

## Even Better Implementation

- Reuse same auxiliary array of size  $n$  for every merge

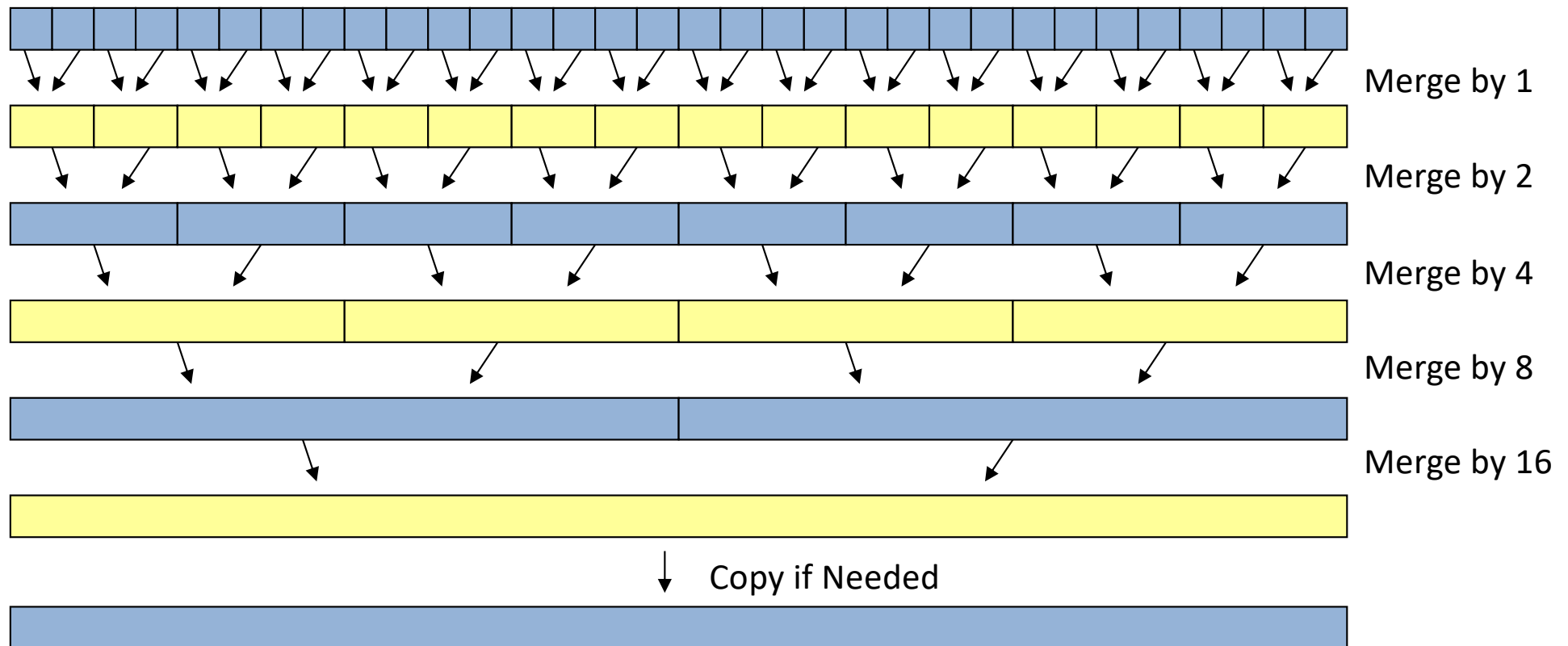
## Best Implementation:

- Do not copy back after merge
- Swap usage of the original and auxiliary array (i.e., even levels move to auxiliary array, odd levels move back to original array)
- Will need one copy at end if number of stages is odd

# Swapping Original & Auxiliary Array

First recurse down to lists of size 1

As we return from the recursion, swap between arrays



Arguably easier to code without using recursion at all

# Mergesort Analysis

Can be made stable and in-place (complex!)

Performance:

To sort  $n$  elements, we

- Return immediately if  $n=1$
- Else do 2 subproblems of size  $n/2$  and then an  $O(n)$  merge
- Recurrence relation:

$$T(1) = c_1$$

$$T(n) = 2T(n/2) + c_2n$$



# MergeSort Recurrence

For simplicity let constants be 1, no effect on asymptotic answer

$$T(1) = 1$$

$$T(n) = 2T(n/2) + n$$

$$= 2(2T(n/4) + n/2) + n$$

$$= 4T(n/4) + 2n$$

$$= 4(2T(n/8) + n/4) + 2n$$

$$= 8T(n/8) + 3n$$

... (after k expansions)

$$= 2^k T(n/2^k) + kn$$

So total is  $2^k T(n/2^k) + kn$   
where  $n/2^k = 1$ , i.e.,  $\log n = k$

That is,  $2^{\log n} T(1) + n \log n$

$$= n + n \log n$$

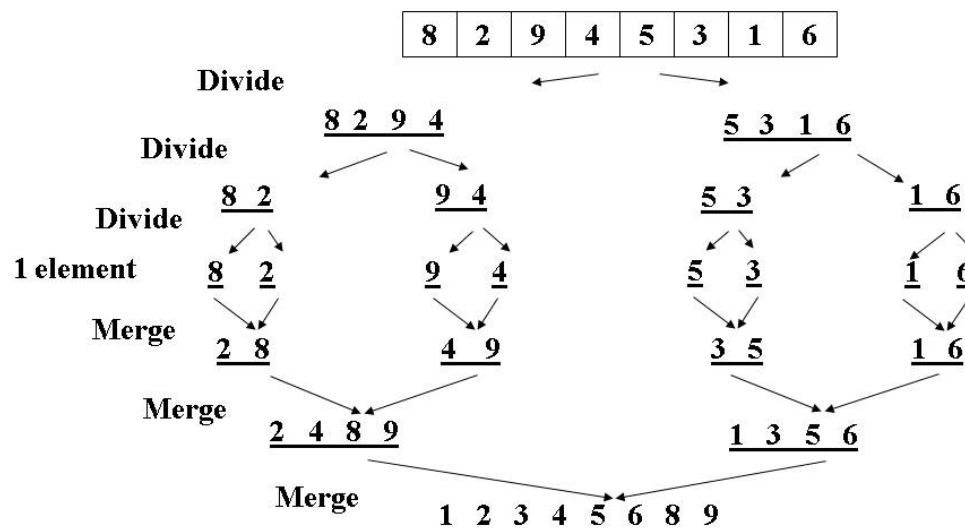
$$= O(n \log n)$$

# Mergesort Analysis

This recurrence is common enough you just “know” it’s  $O(n \log n)$

Merge sort is relatively easy to intuit (best, worst, and average):

- The recursion “tree” will have  $\log n$  height
- At each level we do a total amount of merging equal to  $n$



# Quicksort

Also uses divide-and-conquer

- Recursively chop into halves
- Instead of doing all the work as we merge together, we will do all the work as we recursively split into halves
- Unlike MergeSort, does not need auxiliary space

$O(n \log n)$  on average, but  $O(n^2)$  worst-case

- MergeSort is always  $O(n \log n)$
- So why use QuickSort at all?

Can be faster than Mergesort

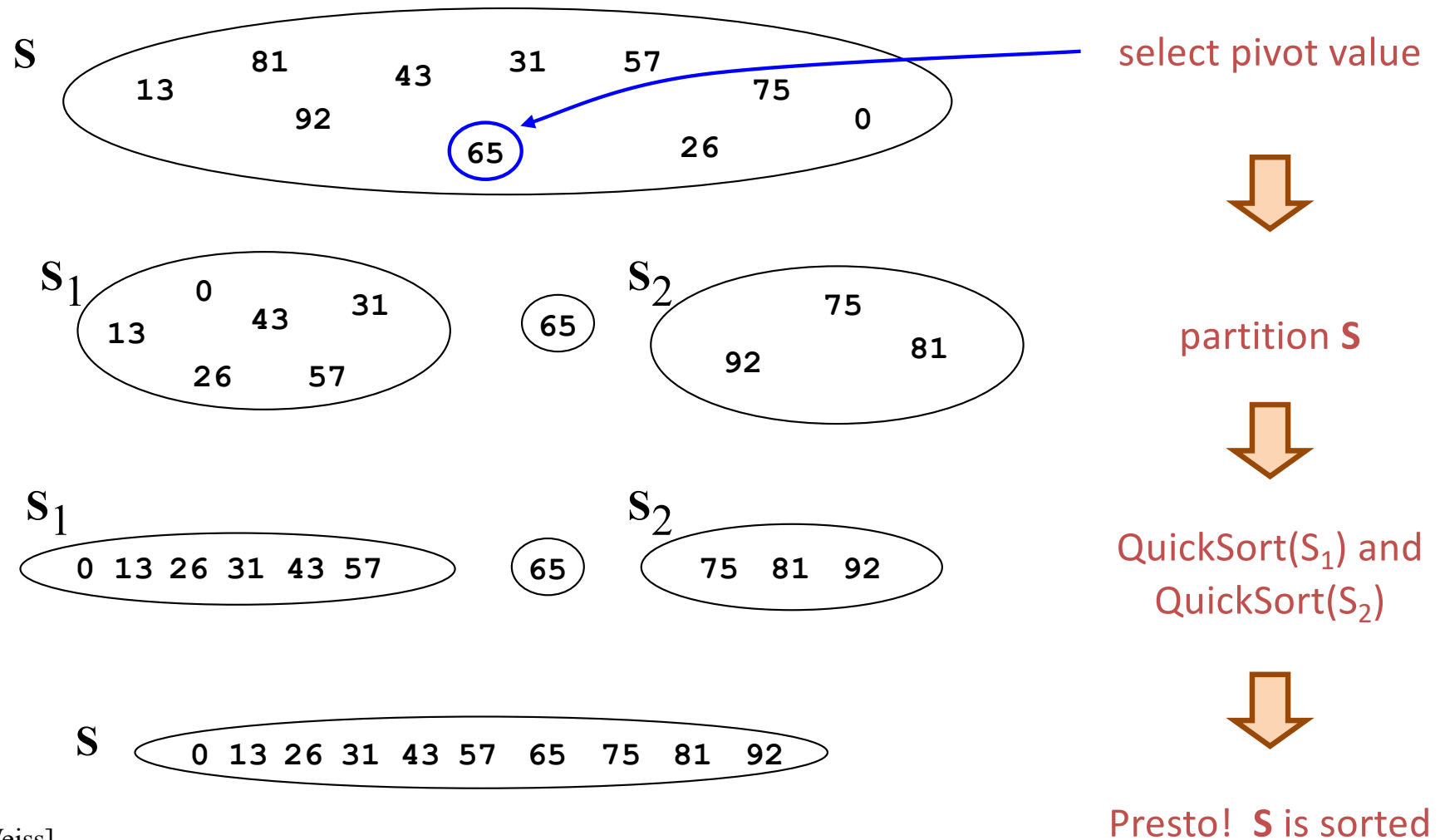
- Believed by many to be faster
- Quicksort does fewer copies and more comparisons, so it depends on the relative cost of these two operations!

# Quicksort Overview

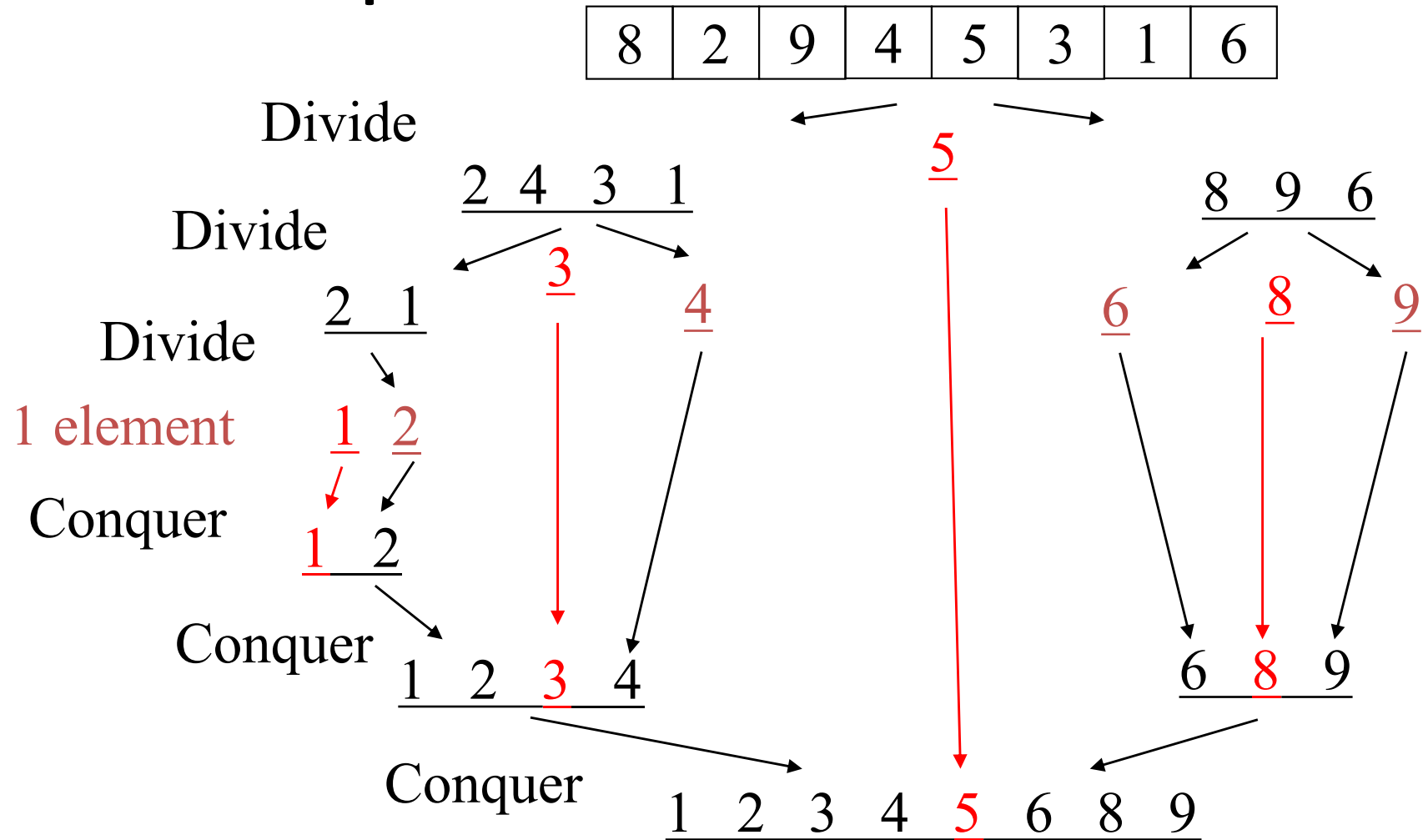
1. Pick a pivot element
2. Partition all the data into:
  - A. The elements less than the pivot
  - B. The pivot
  - C. The elements greater than the pivot
3. Recursively sort A and C
4. The answer is as simple as “A, B, C”

Seems easy by the details are tricky!

# Quicksort: Think in Terms of Sets



# Example: Quicksort Recursion



# Quicksort Details

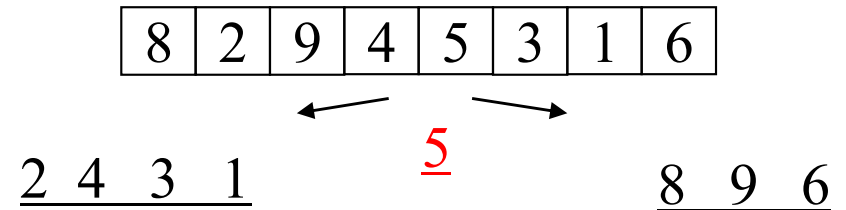
We have not explained:

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

# Pivots

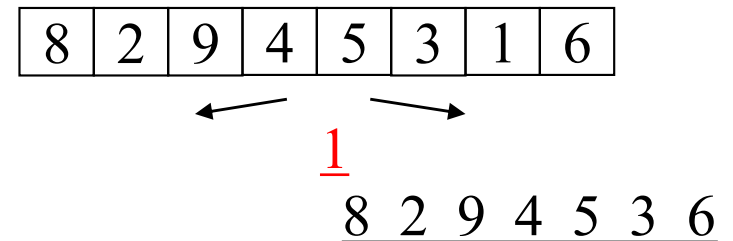
- Best pivot?

- Median
- Halve each time



- Worst pivot?

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





# Quicksort: Potential Pivot Rules

When working on range  $\text{arr}[\text{lo}]$  to  $\text{arr}[\text{hi}-1]$

Pick  $\text{arr}[\text{lo}]$  or  $\text{arr}[\text{hi}-1]$

- Fast but worst-case occurs with nearly sorted input

Pick random element in the range

- Does as well as any technique
- But random number generation can be slow
- Still probably the most elegant approach

Determine median of entire range

- Takes  $O(n)$  time!

Median of 3, (e.g.,  $\text{arr}[\text{lo}]$ ,  $\text{arr}[\text{hi}-1]$ ,  $\text{arr}[(\text{hi}+\text{lo})/2]$ )

- Common heuristic that tends to work well

# Partitioning

Conceptually easy, but hard to correctly code

- Need to partition in linear time *in-place*

One approach (there are slightly fancier ones):

Swap pivot with arr[lo]

Use two fingers i and j, starting at lo+1 and hi-1

while (i < j)

if (arr[j] >= pivot) j--

else if (arr[i] <= pivot) i++

else swap arr[i] with arr[j]

Swap pivot with arr[i]

# Quicksort Example

Step One:

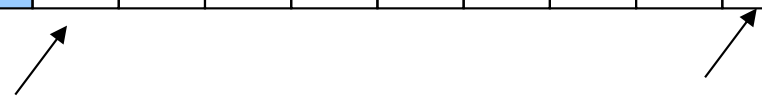
Pick Pivot as Median of 3

lo = 0, hi = 10

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

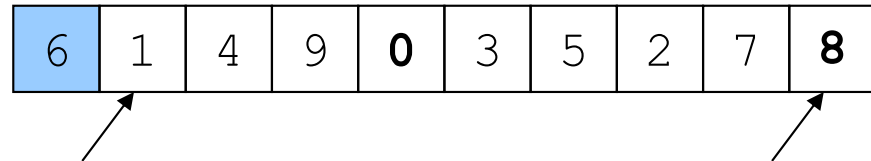
Step Two: Move Pivot to the lo Position

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

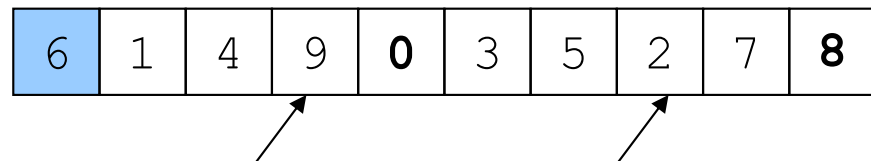


# Quicksort Example

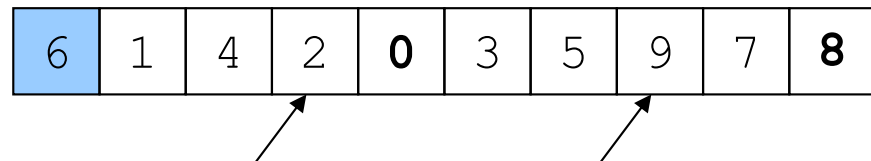
Now partition in place



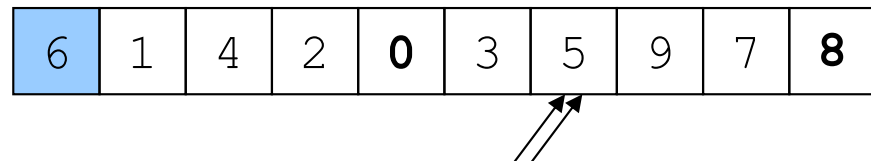
Move fingers



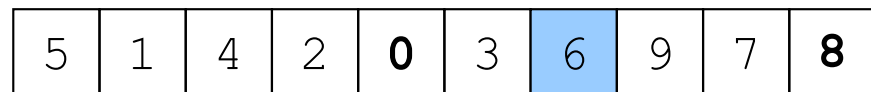
Swap



Move fingers



Move pivot



This is a short example—you typically have more than one swap during partition

# Quicksort Analysis

Best-case: Pivot is always the median

$$T(0)=T(1)=1$$

$$T(n)=2T(n/2) + n \quad \text{linear-time partition}$$

Same recurrence as Mergesort:  $O(n \log n)$

Worst-case: Pivot is always smallest or largest

$$T(0)=T(1)=1$$

$$T(n) = 1T(n-1) + n$$

Basically same recurrence as Selection Sort:  $O(n^2)$

Average-case (e.g., with random pivot):

$$O(n \log n) \text{ (see text)}$$

# Quicksort Cutoffs

For small  $n$ , recursion tends to cost more than a quadratic sort

- Remember asymptotic complexity is for *large*  $n$
- Recursive calls add a lot of overhead for small  $n$

Common technique: switch algorithm below a cutoff

- Rule of thumb: use insertion sort for  $n < 20$

Notes:

- Could also use a cutoff for merge sort
- Cutoffs are also the norm with parallel algorithms (Switch to a sequential algorithm)
- None of this affects asymptotic complexity, just real-world performance

# Quicksort Cutoff Skeleton

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

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
- Smaller arrays are more likely to be nearly sorted

# Linked Lists and Big Data

Mergesort can very nicely work directly on linked lists

- Heapsort and Quicksort do not
- InsertionSort and SelectionSort can too but slower

Mergesort also the sort of choice for external sorting

- Quicksort and Heapsort jump all over the array
- Mergesort scans linearly through arrays
- In-memory sorting of blocks can be combined with larger sorts
- Mergesort can leverage multiple disks



# Sorting: The Big Picture

Horrible  
algorithms:  
 $\Omega(n^2)$

Bogo Sort  
Stooge Sort

Simple  
algorithms:  
 $O(n^2)$

Insertion sort  
Selection sort  
Bubble Sort

Shell sort

...

Fancier  
algorithms:  
 $O(n \log n)$

Heap sort  
Merge sort  
Quick sort (avg)

...

Comparison  
lower bound:  
 $\Omega(n \log n)$

Specialized  
algorithms:  
 $O(n)$

Bucket sort  
Radix sort

# How Fast can we Sort?

Heapsort & Mergesort have  $O(n \log n)$  worst-case run time

Quicksort has  $O(n \log n)$  average-case run time

These bounds are all tight, actually  $\Theta(n \log n)$

So maybe we can dream up another algorithm with a lower asymptotic complexity, such as  $O(n)$  or  $O(n \log \log n)$

- This is unfortunately **IMPOSSIBLE!**
- But why?

# Permutations

Assume we have  $n$  elements to sort

- For simplicity, also assume none are equal (i.e., no duplicates)
- How many permutations of the elements (possible orderings)?

Example,  $n=3$

$a[0] < a[1] < a[2]$	$a[0] < a[2] < a[1]$	$a[1] < a[0] < a[2]$
$a[1] < a[2] < a[0]$	$a[2] < a[0] < a[1]$	$a[2] < a[1] < a[0]$

In general,  $n$  choices for first,  $n-1$  for next,  $n-2$  for next, etc.  $\rightarrow$   
 $n(n-1)(n-2)\dots(1) = n!$  possible orderings

# Representing Every Comparison Sort

Algorithm must “find” the right answer among  $n!$  possible answers

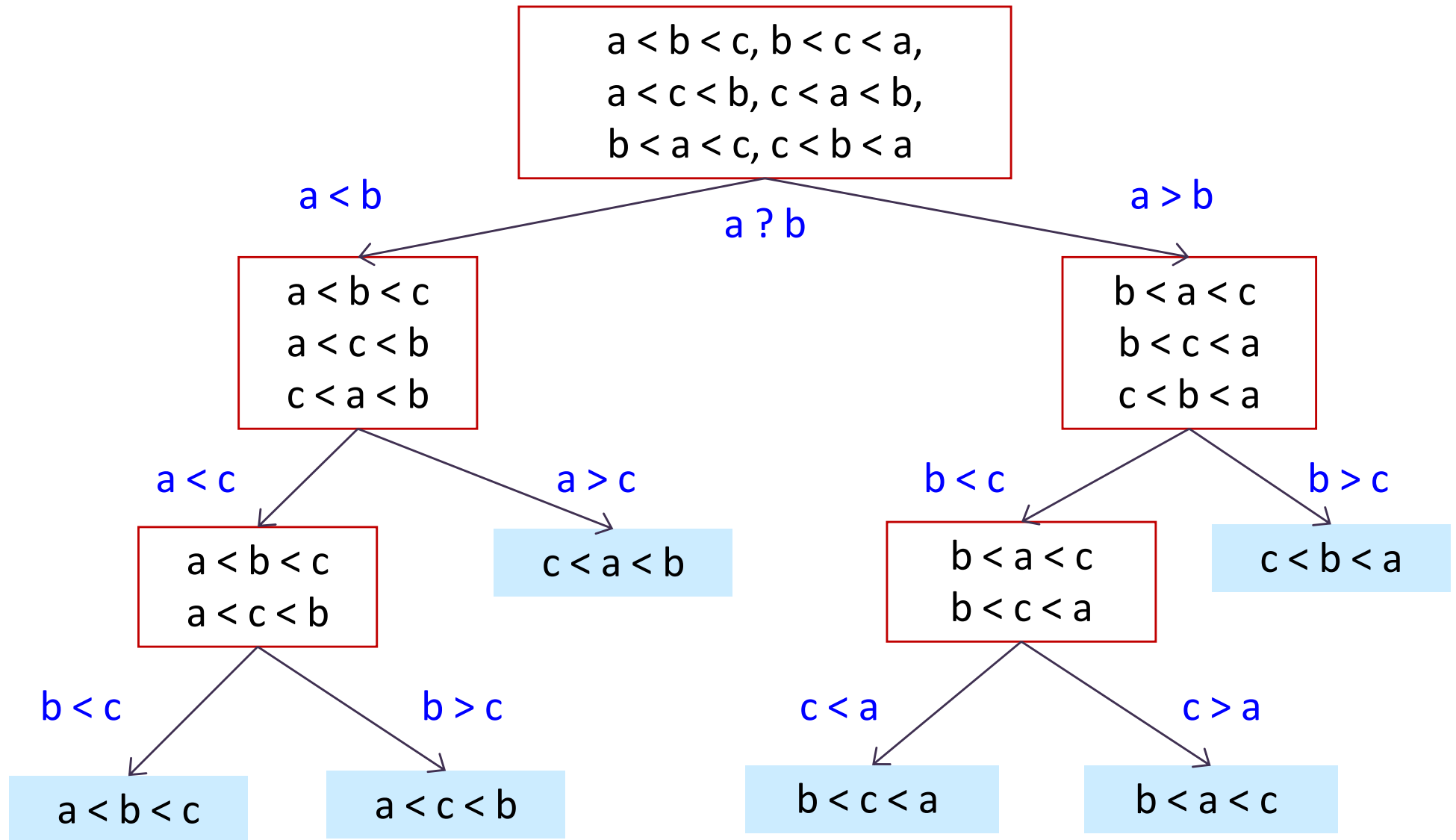
Starts “knowing nothing” and gains information with each comparison

- Intuition is that each comparison can, at best, eliminate half of the remaining possibilities

Can represent this process as a decision tree

- Nodes contain “remaining possibilities”
- Edges are “answers from a comparison”
- This is not a data structure but what our proof uses to represent “the most any algorithm could know”

# Decision Tree for $n = 3$



The leaves contain all the possible orderings of  $a$ ,  $b$ ,  $c$

# What the Decision Tree Tells Us

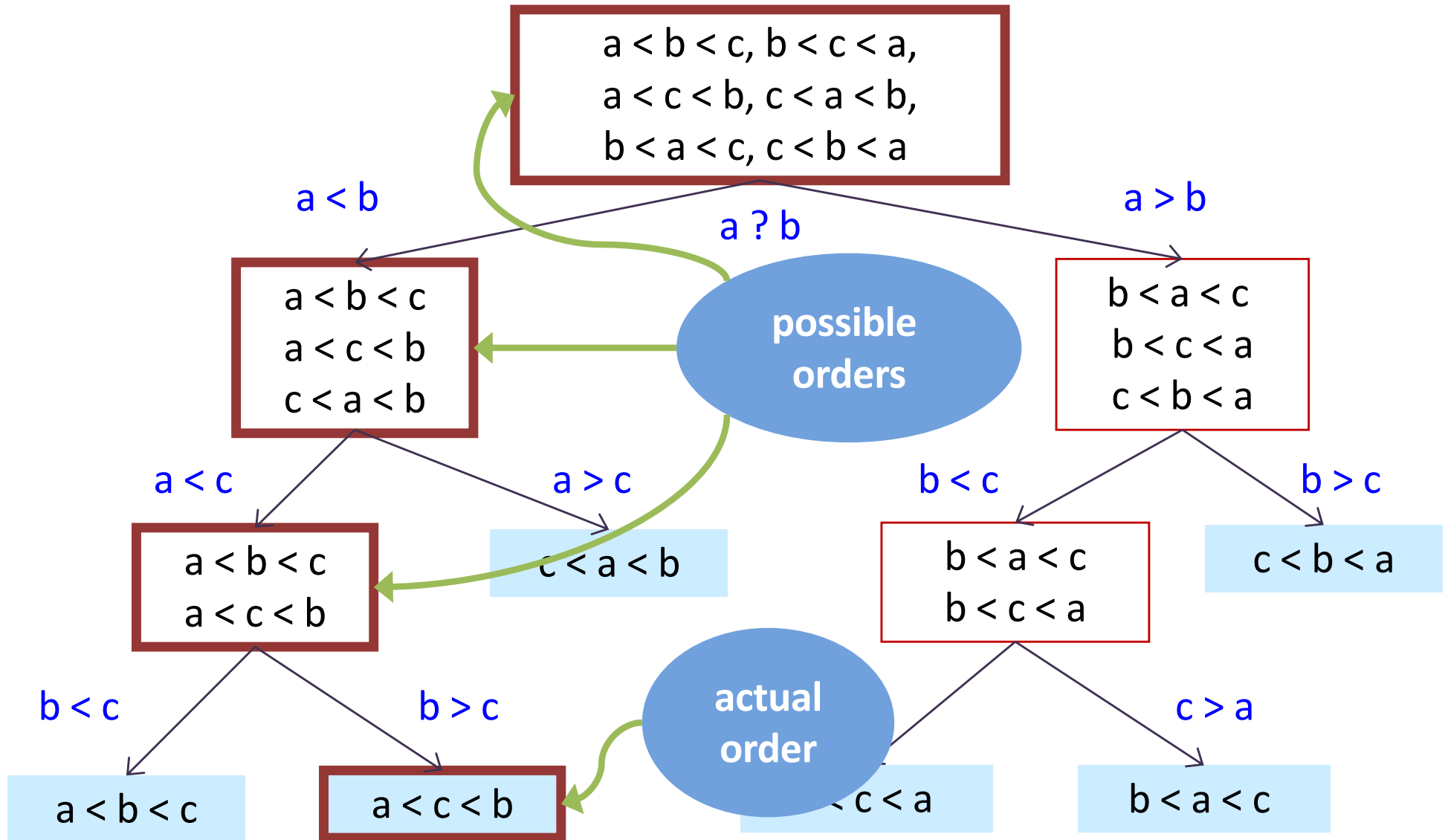
Is a binary tree because

- Each comparison has 2 outcomes
- There are no duplicate elements
- Assumes algorithm does not ask redundant questions

Because any data is possible, any algorithm needs to ask enough questions to decide among all  $n!$  answers

- Every answer is a leaf (no more questions to ask)
- So the tree must be big enough to have  $n!$  leaves
- Running any algorithm on any input will at best correspond to one root-to-leaf path in the decision tree
- So no algorithm can have worst-case running time better than the height of the decision tree

# Decision Tree for $n = 3$



# Where are We

**Proven:** No comparison sort can have worst-case better than the height of a binary tree with  $n!$  leaves

- Turns out average-case is same asymptotically
- So how tall is a binary tree with  $n!$  leaves?

**Now:** Show a binary tree with  $n!$  leaves has height  $\Omega(n \log n)$

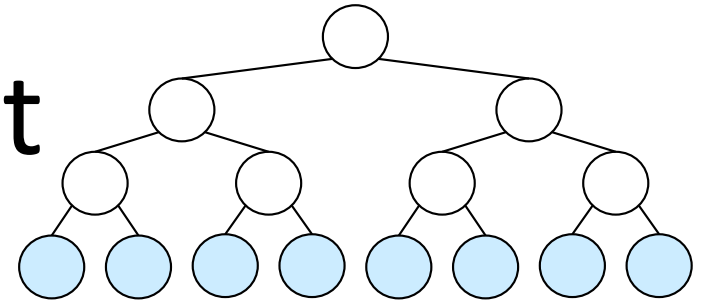
- $n \log n$  is the lower bound, the height must be at least this
- It could be more (in other words, a comparison sorting algorithm could take longer but can not be faster)
- Factorial function grows very quickly

**Conclude that:** (Comparison) Sorting is  $\Omega(n \log n)$

- This is an amazing computer-science result: proves all the clever programming in the world can't sort in linear time!



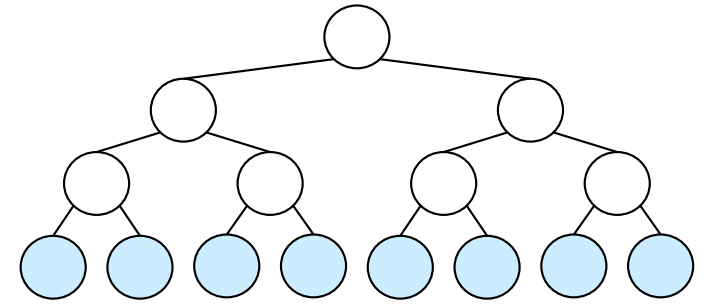
# Lower Bound on Height



- The height of a binary tree with  $L$  leaves is at least  $\log_2 L$
- So the height of our decision tree,  $h$ :

$$\begin{aligned}
 h &\geq \log_2 (n!) && \text{property of binary trees} \\
 &= \log_2 (n \cdot (n-1) \cdot (n-2) \cdots (2)(1)) && \text{definition of factorial} \\
 &= \log_2 n + \log_2 (n-1) + \dots + \log_2 1 && \text{property of logarithms} \\
 &\geq \log_2 n + \log_2 (n-1) + \dots + \log_2 (n/2) && \text{keep first } n/2 \text{ terms} \\
 &\geq (n/2) \log_2 (n/2) && \text{each of the } n/2 \text{ terms left is } \geq \log_2 (n/2) \\
 &\geq (n/2)(\log_2 n - \log_2 2) && \text{property of logarithms} \\
 &\geq (1/2)n \log_2 n - (1/2)n && \text{arithmetic} \\
 &= \Omega(n \log n)
 \end{aligned}$$

# Lower Bound on Height



The height of a binary tree with  $L$  leaves is at least  $\log_2 L$

So the height of our decision tree,  $h$ :

$$\begin{aligned} h &\geq \log_2 (n!) \\ &= \log_2 (n \cdot (n-1) \cdot (n-2) \cdots (2)(1)) \\ &= \log_2 n + \log_2 (n-1) + \dots + \log_2 1 \\ &\geq \log_2 n + \log_2 (n-1) + \dots + \log_2 (n/2) \\ &\geq (n/2) \log_2 (n/2) \\ &= (n/2)(\log_2 n - \log_2 2) \\ &\geq (1/2)n \log_2 n - (1/2)n \\ &=" \Omega(n \log n) \end{aligned}$$

# **BREAKING THE $\Omega(N \log N)$ BARRIER FOR SORTING**

# Sorting: The Big Picture

Horrible  
algorithms:  
 $\Omega(n^2)$

Bogo Sort  
Stooge Sort

Simple  
algorithms:  
 $O(n^2)$

Insertion sort  
Selection sort  
Bubble Sort

Shell sort

...

Fancier  
algorithms:  
 $O(n \log n)$

Heap sort  
Merge sort  
Quick sort (avg)

...

Comparison  
lower bound:  
 $\Omega(n \log n)$

Specialized  
algorithms:  
 $O(n)$

Bucket sort  
Radix sort

# BucketSort (a.k.a. BinSort)

If all values to be sorted are known to be integers between 1 and K (or any small range),

- Create an array of size K

- Put each element in its proper **bucket (a.k.a. bin)**

- If data is only integers, only need to store the count of how times that bucket has been used

Output result via linear pass through array of buckets

count array	
1	
2	
3	
4	
5	

Example:

K=5

Input: (5, 1, 3, 4, 3, 2, 1, 1, 5, 4, 5)

Output:

# BucketSort (a.k.a. BinSort)

If all values to be sorted are known to be integers between 1 and K (or any small range),

Create an array of size K

Put each element in its proper **bucket (a.k.a. bin)**

If data is only integers, only need to store the count of how times that bucket has been used

Output result via linear pass through array of buckets

count array	
1	3
2	1
3	2
4	2
5	3

Example:

K=5

Input: (5, 1, 3, 4, 3, 2, 1, 1, 5, 4, 5)

Output:

# BucketSort (a.k.a. BinSort)

If all values to be sorted are known to be integers between 1 and K (or any small range),

Create an array of size K

Put each element in its proper **bucket (a.k.a. bin)**

If data is only integers, only need to store the count of how times that bucket has been used

Output result via linear pass through array of buckets

count array	
1	3
2	1
3	2
4	2
5	3

Example:

K = 5

Input: (5, 1, 3, 4, 3, 2, 1, 1, 5, 4, 5)

Output: (1, 1, 1, 2, 3, 3, 4, 4, 5, 5, 5)

What is the running time?

# Analyzing Bucket Sort

Overall:  $O(n+K)$

- Linear in  $n$ , but also linear in  $K$
- $\Omega(n \log n)$  lower bound does not apply because this is not a comparison sort

Good when  $K$  is smaller (or not much larger) than  $n$

- Do not spend time doing comparisons of duplicates

Bad when  $K$  is much larger than  $n$

- Wasted space / time during final linear  $O(K)$  pass



# Bucket Sort with Data

For data in addition to integer keys, use list at each bucket

count array	
1	→ Twilight
2	
3	→ Harry Potter
4	
5	→ Gattaca → Star Wars

Bucket sort illustrates a more general trick

- Imagine a heap for a small range of integer priorities

# Radix Sort (originated 1890 census)

Radix = “the base of a number system”

- Examples will use our familiar base 10
- Other implementations may use larger numbers (e.g., ASCII strings might use 128 or 256)

Idea:

- Bucket sort on one digit at a time
  - Number of buckets = radix
  - Starting with least significant digit, sort with Bucket Sort
  - Keeping sort stable
- Do one pass per digit
- After k passes, the last k digits are sorted

# Example: Radix Sort: Pass #1

Input data

478  
537  
9  
721  
3  
38  
123  
67

Bucket sort by 1's digit

0	1	2	3	4	5	6	7	8	9
	72 <u>1</u>		<u>3</u> 12 <u>3</u>				53 <u>7</u> 6 <u>7</u>	47 <u>8</u> 3 <u>8</u>	<u>9</u>

After 1<sup>st</sup> pass

721  
3  
123  
537  
67  
478  
38  
9

This example uses B=10 and base 10 digits for simplicity of demonstration. Larger bucket counts should be used in an actual implementation.

# Example: Radix Sort: Pass #2

After 1<sup>st</sup> pass

721  
3  
123  
537  
67  
478  
38  
9

Bucket sort by 10's digit

0	1	2	3	4	5	6	7	8	9
<u>0</u> 3 <u>0</u> 9		<u>7</u> 21 <u>1</u> 23	<u>5</u> 37 <u>3</u> 8			<u>6</u> 7	<u>4</u> 78		

After 2<sup>nd</sup> pass

3  
9  
721  
123  
537  
38  
67  
478

# Example: Radix Sort: Pass #3

After 2<sup>nd</sup> pass

3  
9  
721  
123  
537  
38  
67  
478

Bucket sort by 10's digit

0	1	2	3	4	5	6	7	8	9
<u>0</u> 03	<u>1</u> 23			<u>4</u> 78	<u>5</u> 37		<u>7</u> 21		
<u>0</u> 09									
<u>0</u> 38									
<u>0</u> 67									

After 3<sup>rd</sup> pass

3  
9  
38  
67  
123  
478  
537  
721

**Invariant:**

After k passes the low order k digits are sorted.

# Analysis

Input size:  $n$

Number of buckets = Radix:  $B$

Number of passes = “Digits”:  $P$

Work per pass is 1 bucket sort:  $O(B + n)$

Total work is  $O(P \cdot (B + n))$

Better/worse than comparison sorts? Depends on  $n$

Example: Strings of English letters up to length 15

- $15 \cdot (52 + n)$
- This is less than  $n \log n$  only if  $n > 33,000$
- Of course, cross-over point depends on constant factors of the implementations

# Sorting Summary

Simple  $O(n^2)$  sorts can be fastest for small  $n$

- Selection sort, Insertion sort (is linear for nearly-sorted)
- Both stable and in-place
- Good for “below a cut-off” to help divide-and-conquer sorts

$O(n \log n)$  sorts

- Heapsort, in-place but not stable nor parallelizable
- Mergesort, not in-place but stable and works as external sort
- Quicksort, in-place but not stable and  $O(n^2)$  in worst-case  
Often fastest, but depends on costs of comparisons/copies

$\Omega(n \log n)$  worst and average bound for comparison sorting

Non-comparison sorts

- Bucket sort good for small number of key values
- Radix sort uses fewer buckets and more phases