

CSE 332: Data Structures & Parallelism

Lecture 12: Comparison Sorting

Ruth Anderson
Winter 2025

Today

- Sorting
 - Comparison sorting

Introduction to sorting

- Stacks, queues, priority queues, and dictionaries all focused on providing one element at a time
- But often we know we want “all the data items” in some order
 - Anyone can sort, but a computer can sort faster
 - Very common to need data sorted somehow
 - Alphabetical list of people
 - Population list of countries
 - Search engine results by relevance
 - ...
- Different algorithms have different asymptotic and constant-factor trade-offs
 - No single ‘best’ sort for all scenarios
 - Knowing one way to sort just isn’t enough

More reasons to sort

General technique in computing:

Preprocess (e.g. sort) data to make subsequent operations faster

Example: Sort the data so that you can

- Find the k^{th} largest in constant time for any k
- Perform binary search to find an element in logarithmic time

Whether the benefit of the preprocessing depends on

- How often the data will change
- How much data there is

The main problem, stated carefully

For now we will assume we have n comparable elements in an array and we want to rearrange them to be in increasing order

Input:

- An array \mathbf{A} of data records
- A key value in each data record
- A comparison function (consistent and total)
 - Given keys a & b , what is their relative ordering? $<$, $=$, $>$?
 - Ex: keys that implement Comparable or have a Comparator that can handle them

Effect:

- Reorganize the elements of \mathbf{A} such that for any i and j ,
if $i < j$ then $\mathbf{A}[i] \leq \mathbf{A}[j]$
- Usually unspoken assumption: \mathbf{A} must have all the same data it started with
- Could also sort in reverse order, of course

An algorithm doing this is a **comparison sort**

Variations on the basic problem

1. Maybe elements are in a linked list (could convert to array and back in linear time, but some algorithms needn't do so)
2. Maybe in the case of ties we should preserve the original ordering
 - Sorts that do this naturally are called **stable sorts**
 - One way to sort twice, Ex: Sort movies by year, then for ties, alphabetically
3. Maybe we must not use more than $O(1)$ “auxiliary space”
 - Sorts meeting this requirement are called ‘**in-place**’ sorts
 - Not allowed to allocate extra array (at least not with size $O(n)$), but can allocate $O(1)$ # of variables
 - All work done by swapping around in the array
4. Maybe we can do more with elements than just compare
 - Comparison sorts assume we work using a binary ‘compare’ operator
 - In special cases we can sometimes get faster algorithms
5. Maybe we have too much data to fit in memory
 - Use an “**external sorting**” algorithm

Sorting: The Big Picture

Simple algorithms:
 $O(n^2)$

Insertion sort
Selection 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

Handling huge data sets

External sorting

Insertion Sort

- Idea: At step k , put the k^{th} element in the correct position among the first k elements
- Alternate way of saying this:
 - Sort first two elements (by swapping them if needed)
 - Now compare 3rd element to 2nd (swap & keep swapping as needed until no longer needs to move)
 - Now compare 4th element to 3rd (swap & keep swapping as needed until no longer needs to move)
 - ...
- “Loop invariant”: when loop index is i , first i elements are sorted relative to each other
- Time?
Best-case _____ Worst-case _____ “Average” case _____

Selection sort

- Idea: At step k , find the smallest element among the not-yet-sorted elements and put it at position k
- Alternate way of saying this:
 - Find smallest element, swap it into 1st location
 - Find next smallest element, swap it into 2nd location
 - Find next smallest element, swap it into 3rd location
 - ...
- “Loop invariant”: when loop index is i , first i elements are the i smallest elements in sorted order
- Time?
Best-case _____ Worst-case _____ “Average” case _____

Insertion Sort vs. Selection Sort

- Different algorithms
- Solve the same problem
- Have the same worst-case and average-case asymptotic complexity
 - Insertion-sort has better best-case complexity; preferable when input is “mostly sorted”
- Other algorithms are more efficient *for non-small arrays that are not already almost sorted*
 - Insertion sort may do well on small arrays

Aside: We won't cover Bubble Sort

- It doesn't have good asymptotic complexity: $O(n^2)$
- It's not particularly efficient with respect to common factors
- Basically, almost everything it is good at, some other algorithm is at least as good at
- Some people seem to teach it just because someone taught it to them
- For fun see: "Bubble Sort: An Archaeological Algorithmic Analysis", Owen Astrachan, SIGCSE 2003
<http://www.cs.duke.edu/~ola/bubble/bubble.pdf>

Sorting: The Big Picture

Simple algorithms:
 $O(n^2)$

Insertion sort
Selection 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

Handling huge data sets

External sorting

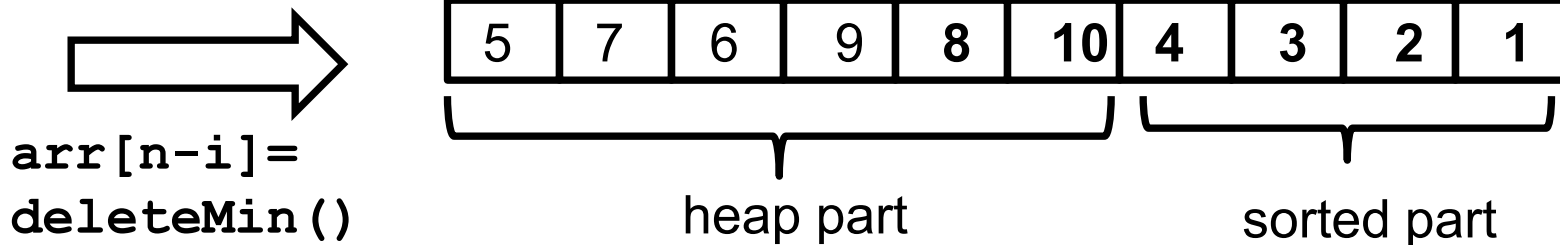
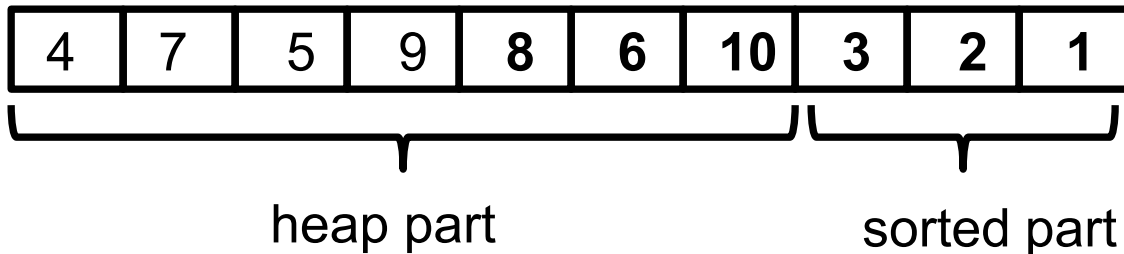
Heap sort

- Sorting with a heap is easy:
 - `insert` each `arr[i]`, better yet use `buildHeap`
 - `for(i=0; i < arr.length; i++)`
`arr[i] = deleteMin();`
- Worst-case running time:
- We have the array-to-sort and the heap
 - So this is not an in-place sort
 - There's a trick to make it in-place...

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^{th} element, put it at `arr[n-i]`
 - It's not part of the heap anymore!



“AVL sort”

- How?

Divide and conquer

Very important technique in algorithm design

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

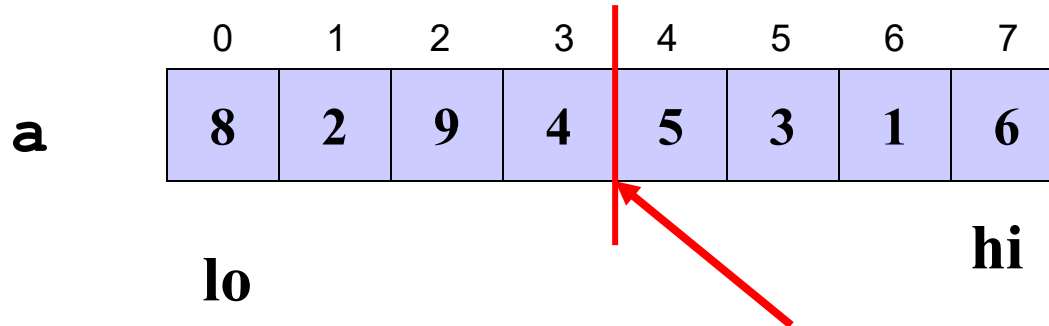
Ex: Sort each half of the array, combine together; to sort each half, split into halves...

Divide-and-conquer sorting

Two great sorting methods are fundamentally divide-and-conquer

1. 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
2. Quicksort: Pick a “pivot” element
Divide elements into those less-than pivot
and those greater-than pivot
Sort the two divisions (recursively on each)
Answer is [*sorted-less-than* then *pivot* then
sorted-greater-than]

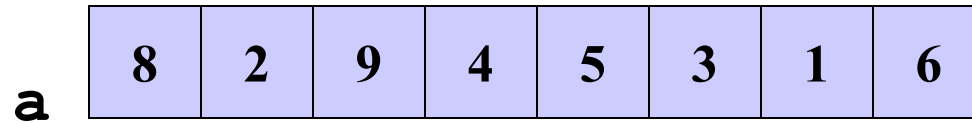
Mergesort



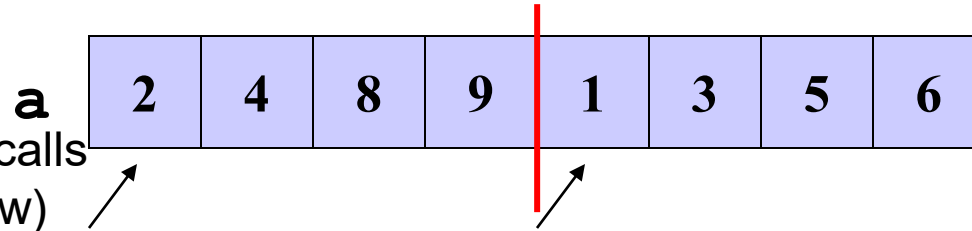
- To sort array from position **lo** to position **hi**:
 - If range is 1 element long, it's sorted! (Base case)
 - Else, split into two halves:
 - Sort from **lo** to $(\mathbf{hi} + \mathbf{lo}) / 2$
 - Sort from $(\mathbf{hi} + \mathbf{lo}) / 2$ to **hi**
 - Merge the two halves together
- Merging takes two sorted parts and sorts everything
 - $O(n)$ but requires auxiliary space...

Example, focus on merging

Start with:

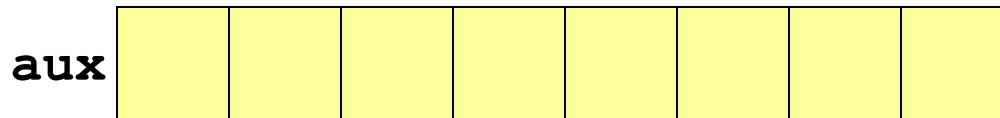


After we return from
left and right recursive calls
(pretend it works for now)



Merge:

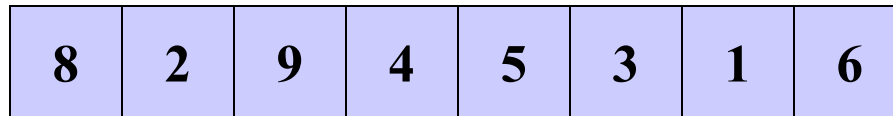
Use 3 “fingers”
and 1 more array



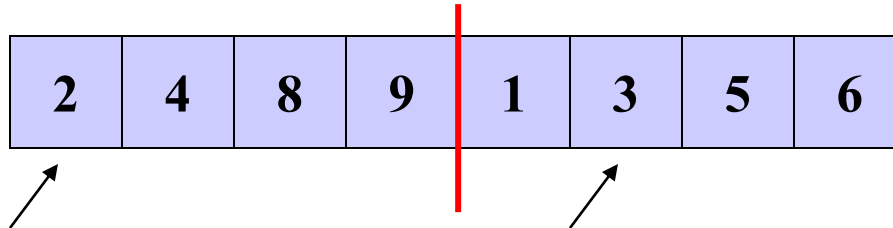
(After merge,
copy back to
original array)

Example, focus on merging

Start with:

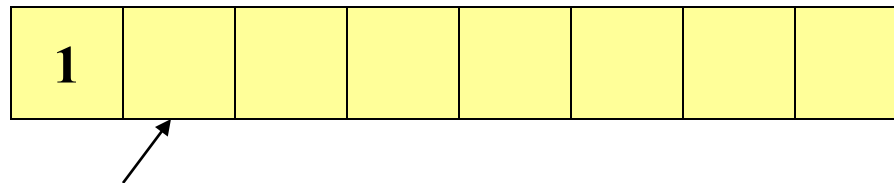


After recursion:
(not magic 😊)



Merge:

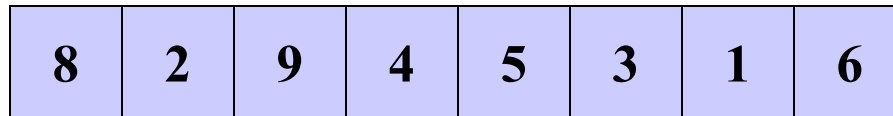
Use 3 “fingers”
and 1 more array



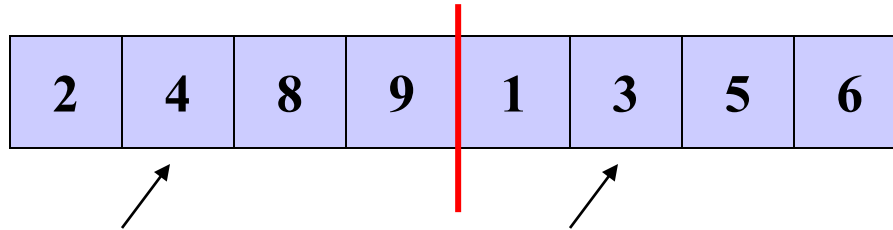
(After merge,
copy back to
original array)

Example, focus on merging

Start with:

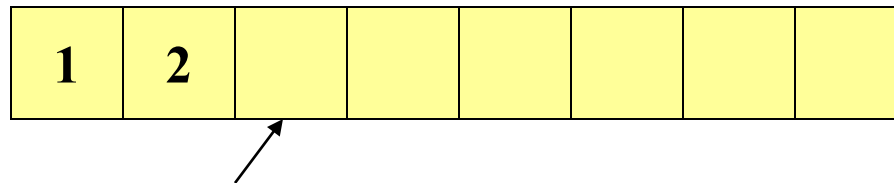


After recursion:
(not magic 😊)



Merge:

Use 3 “fingers”
and 1 more array



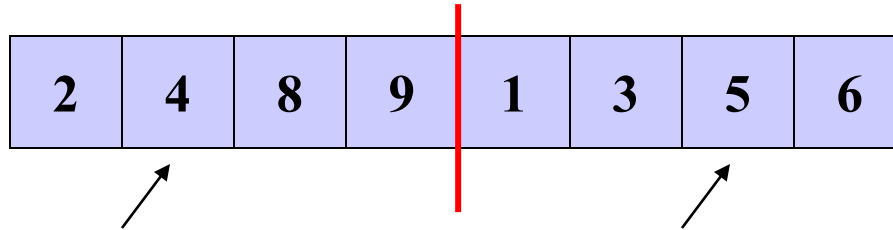
(After merge,
copy back to
original array)

Example, focus on merging

Start with:

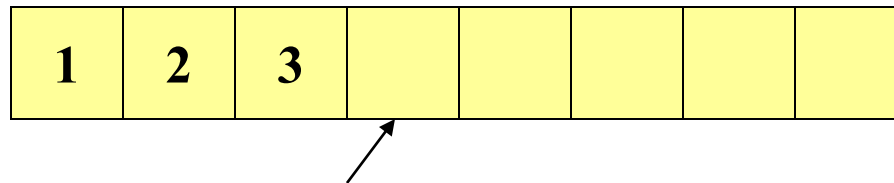


After recursion:
(not magic 😊)



Merge:

Use 3 “fingers”
and 1 more array



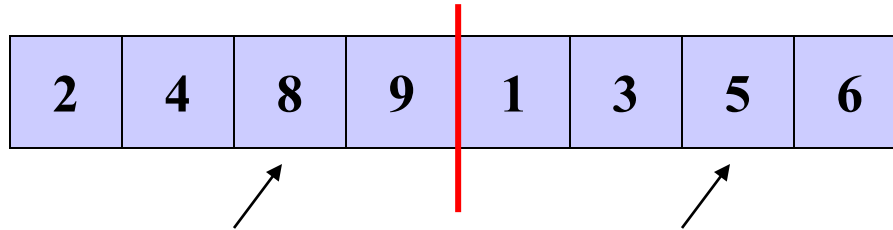
(After merge,
copy back to
original array)

Example, focus on merging

Start with:

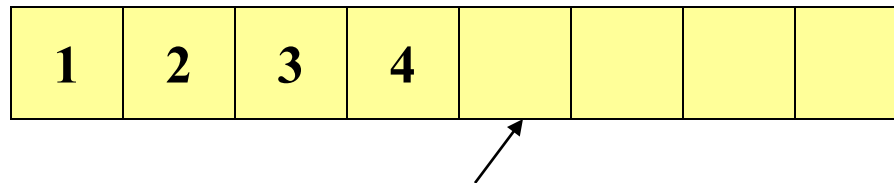


After recursion:
(not magic 😊)



Merge:

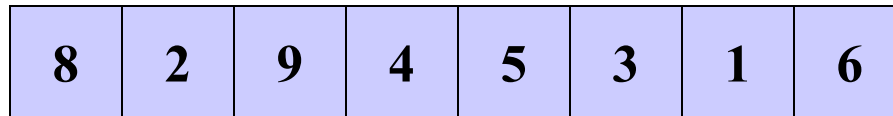
Use 3 “fingers”
and 1 more array



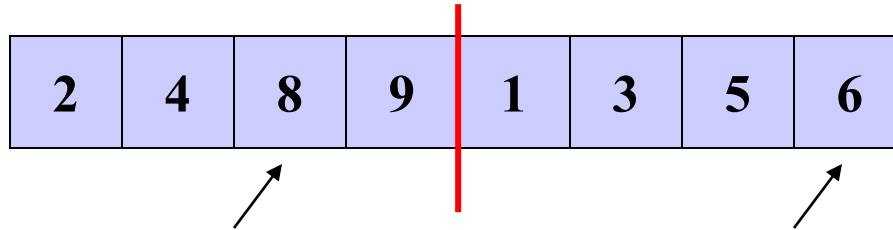
(After merge,
copy back to
original array)

Example, focus on merging

Start with:

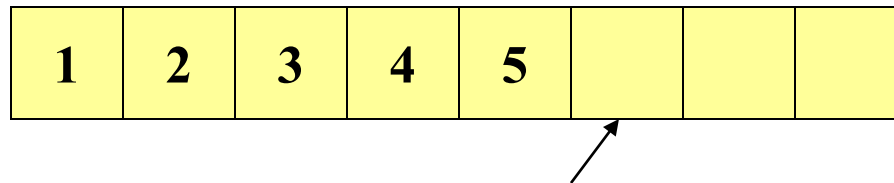


After recursion:
(not magic 😊)



Merge:

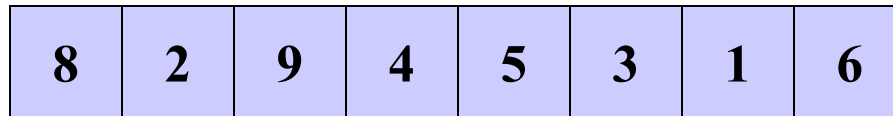
Use 3 “fingers”
and 1 more array



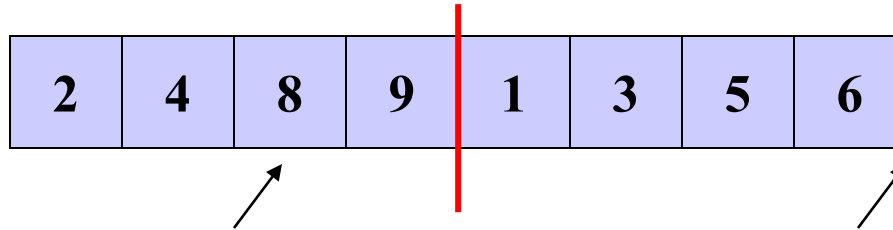
(After merge,
copy back to
original array)

Example, focus on merging

Start with:

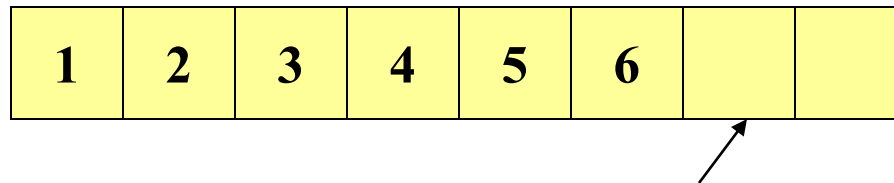


After recursion:
(not magic 😊)



Merge:

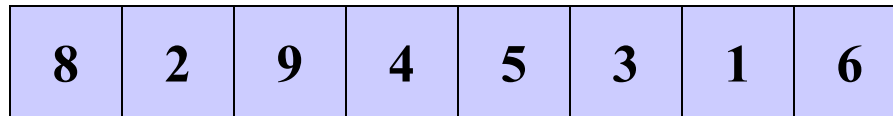
Use 3 “fingers”
and 1 more array



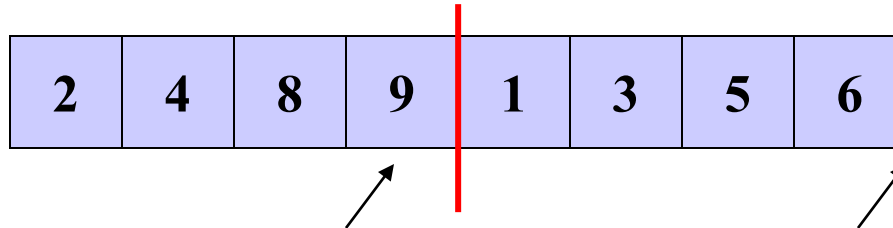
(After merge,
copy back to
original array)

Example, focus on merging

Start with:

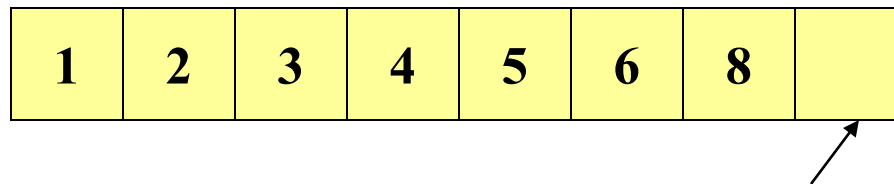


After recursion:
(not magic 😊)



Merge:

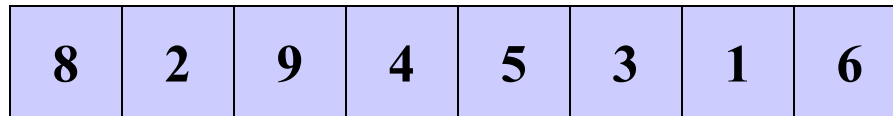
Use 3 “fingers”
and 1 more array



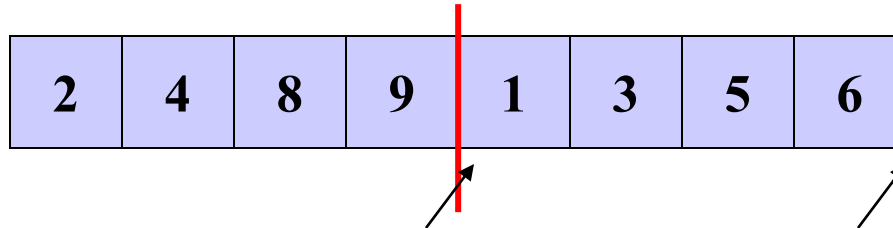
(After merge,
copy back to
original array)

Example, focus on merging

Start with:

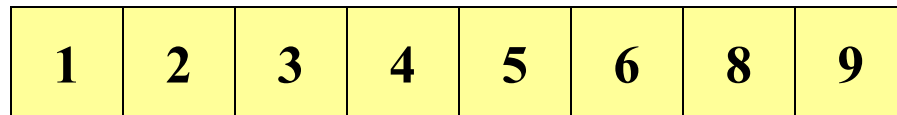


After recursion:
(not magic 😊)



Merge:

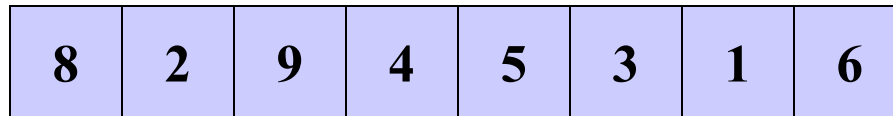
Use 3 “fingers”
and 1 more array



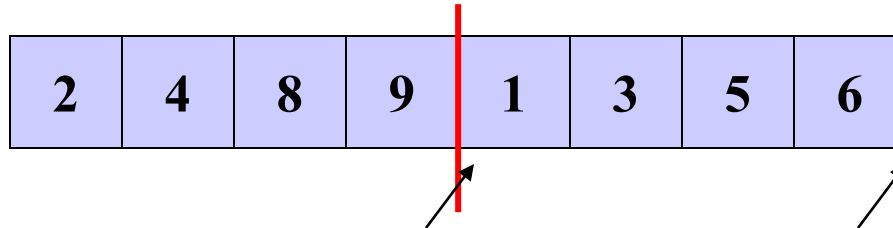
(After merge,
copy back to
original array)

Example, focus on merging

Start with:

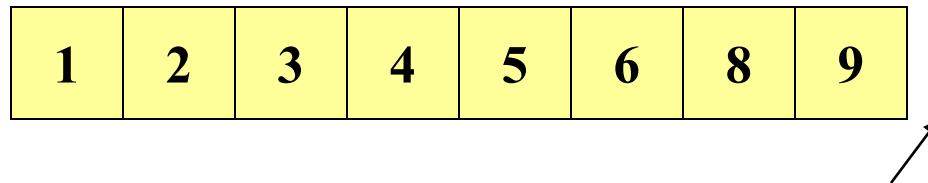


After recursion:
(not magic 😊)

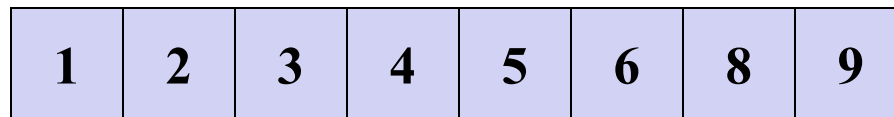


Merge:

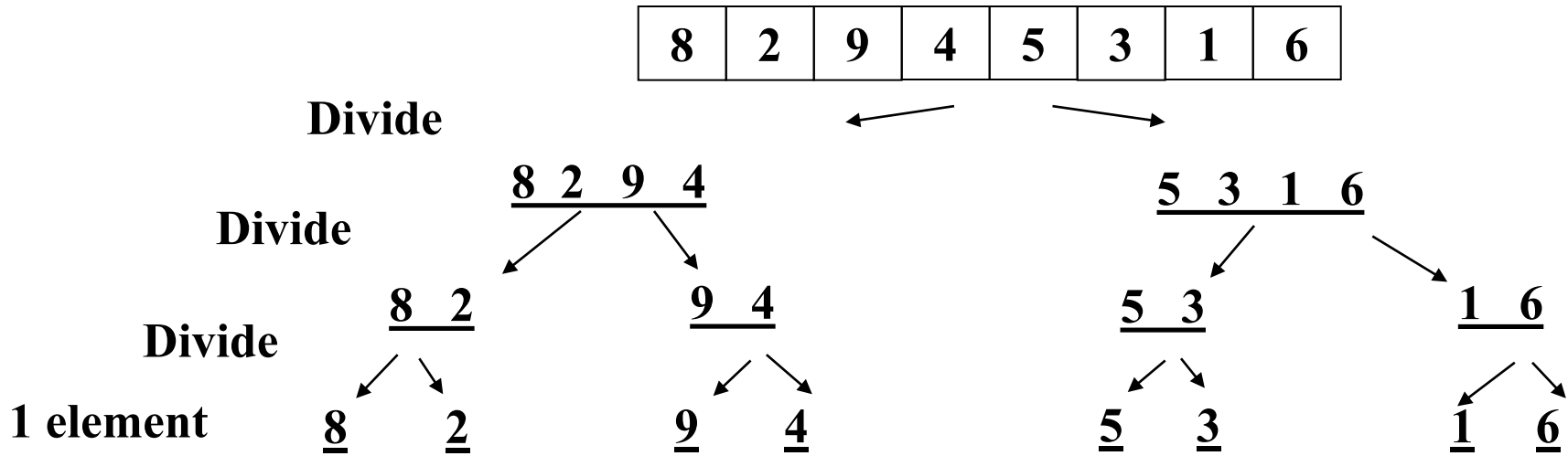
Use 3 “fingers”
and 1 more array



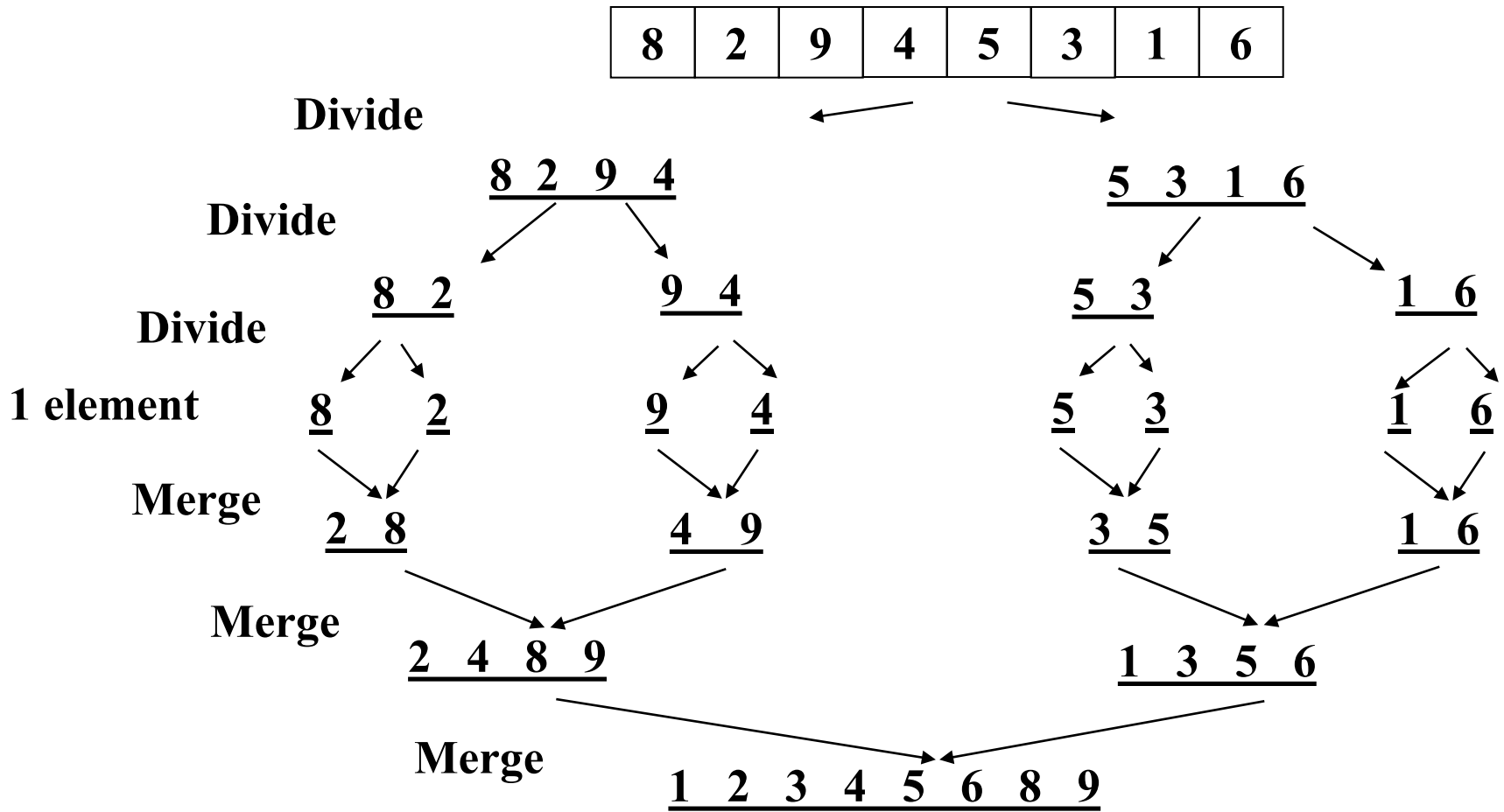
(After merge,
copy back to
original array)



Mergesort example: Recursively splitting list in half

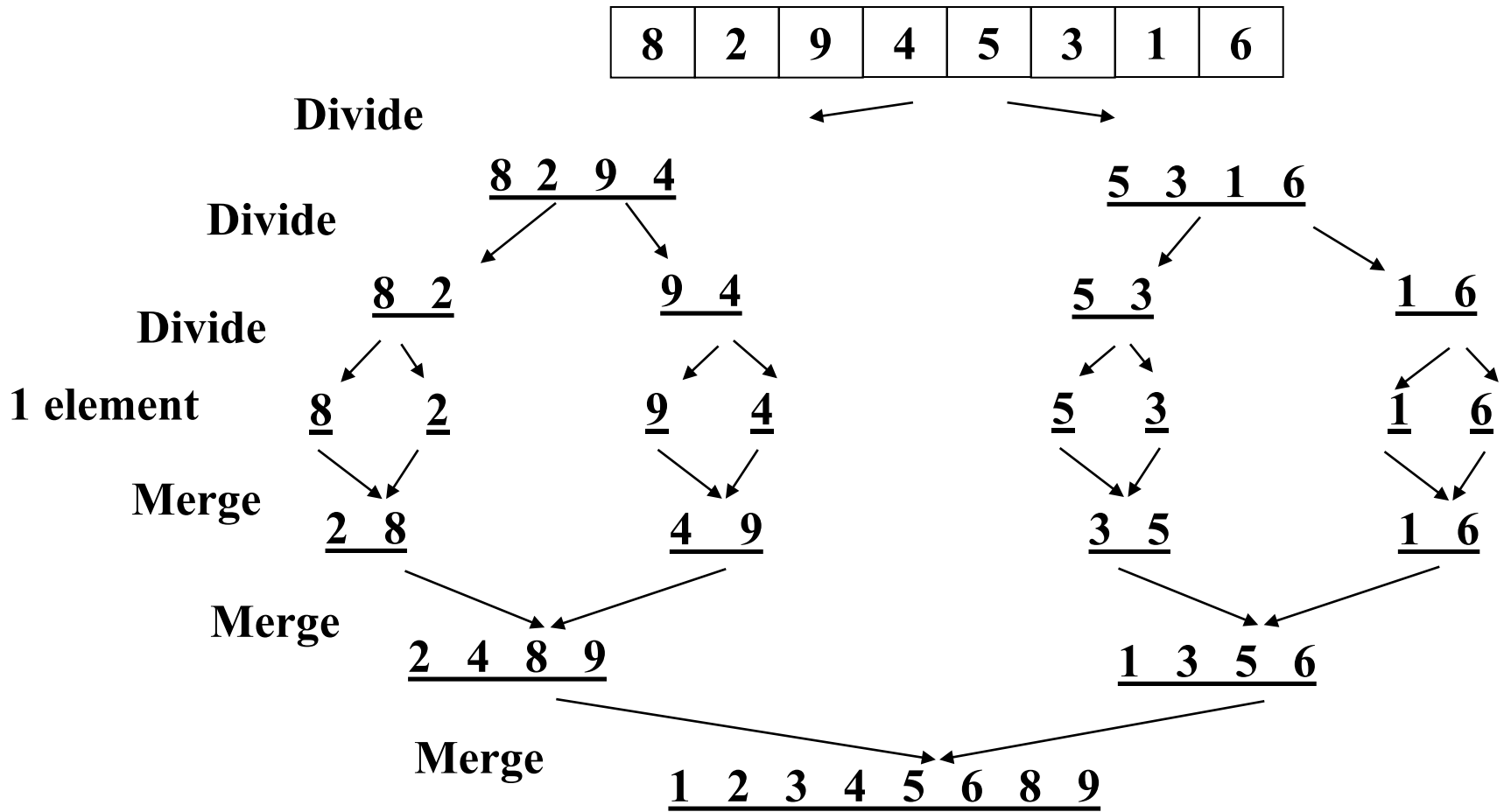


Mergesort example: Merge as we return from recursive calls



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

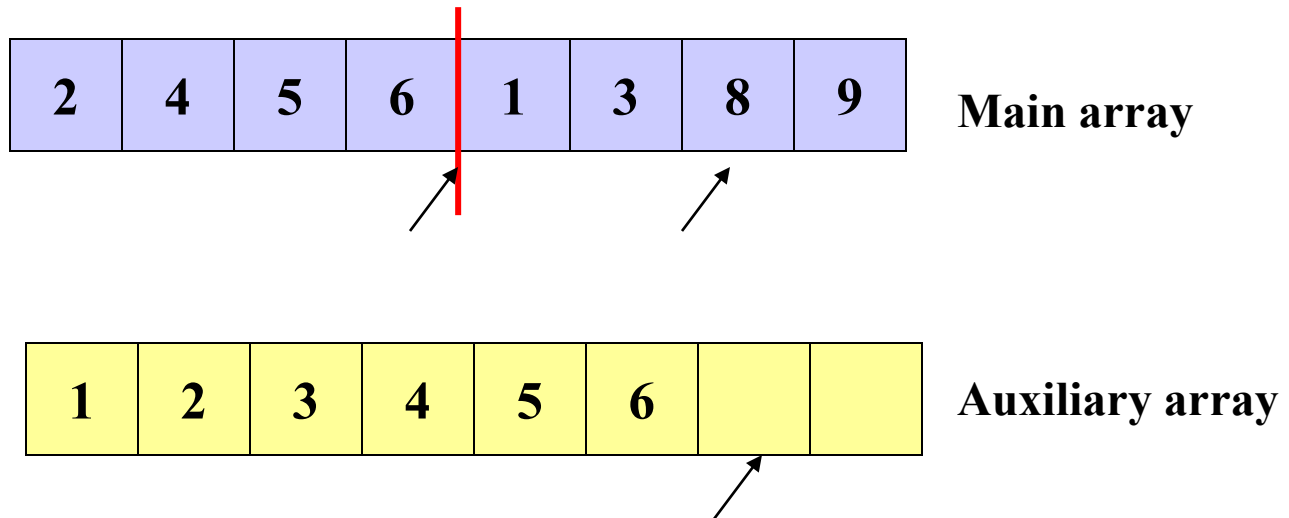
Mergesort example: Merge as we return from recursive calls



We need another array in which to do each merging step; merge results into there, then copy back to original array

Mergesort, some details: saving a little time

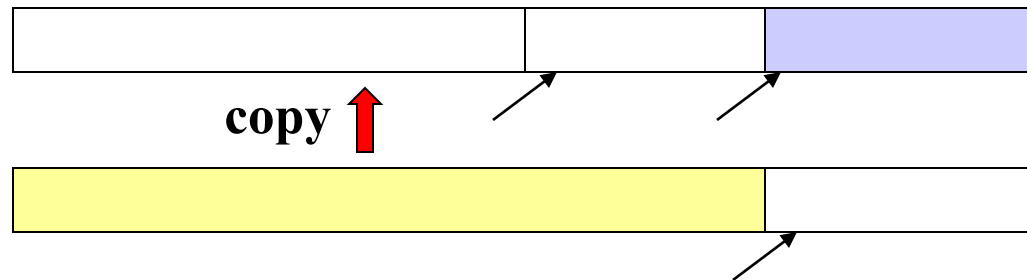
- What if the final steps of our merging looked like the following:



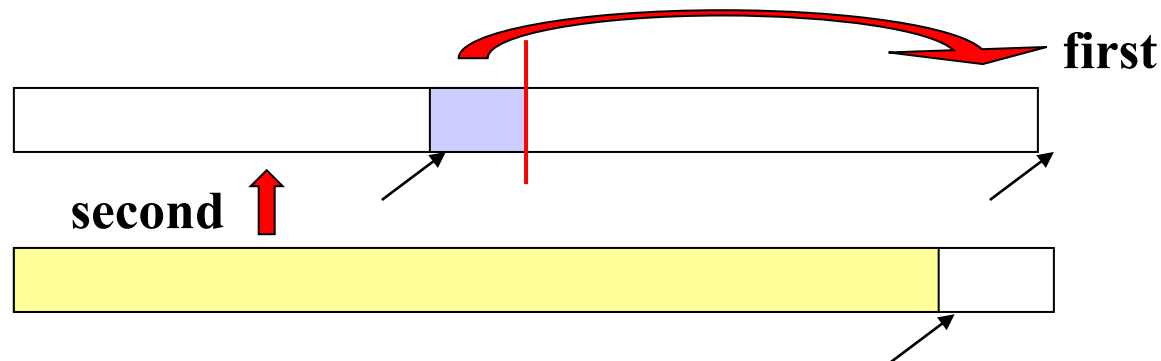
- Seems kind of wasteful to copy 8 & 9 to the auxiliary array just to copy them immediately back...

Mergesort, some details: saving a little time

- Unnecessary to copy 'dregs' over to auxiliary array
 - If left-side finishes first, just stop the merge & copy the auxiliary array:



- If right-side finishes first, copy dregs directly into right side, then copy auxiliary array



Some details: saving space / copying

Simplest / worst approach:

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

Returning from a recursive call? Allocate a new array!

Better:

Reuse same auxiliary array of size n for every merging stage

Allocate auxiliary array at beginning, use throughout

Best (but a little tricky):

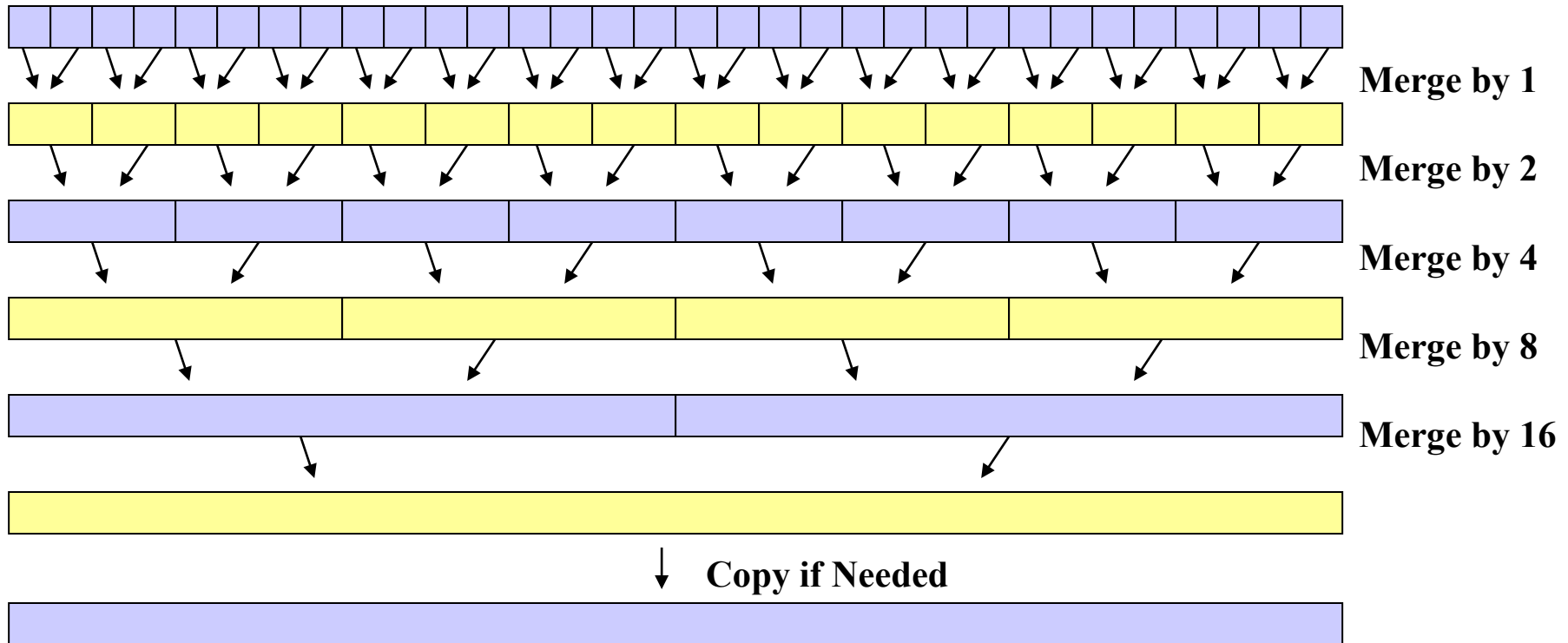
Don't copy back – at 2nd, 4th, 6th, ... merging stages, use the original array as the auxiliary array and vice-versa

– Need one copy at end if number of stages is odd

*Picture of the “best” from previous slide:
Allocate one auxiliary array, switch each step*

First recurse down to lists of size 1

As we return from the recursion, switch off arrays



Arguably easier to code up without recursion at all

Aside: Linked lists and big data

We defined the sorting problem as over an array, but sometimes you want to sort linked lists

One approach:

- Convert to array: $O(n)$
- Sort: $O(n \log n)$
- Convert back to list: $O(n)$

Or: mergesort works very nicely on linked lists directly

- heapsort and quicksort do not
- insertion sort and selection sort do but they're slower

Mergesort is also the sort of choice for external sorting

- Linear merges minimize disk accesses

Mergesort Analysis

Having defined an algorithm and argued it is correct, we should analyze its running time (and space):

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?

Mergesort Analysis

Having defined an algorithm and argued it is correct, we should analyze its running time (and space):

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:

$$T(1) = c_1$$

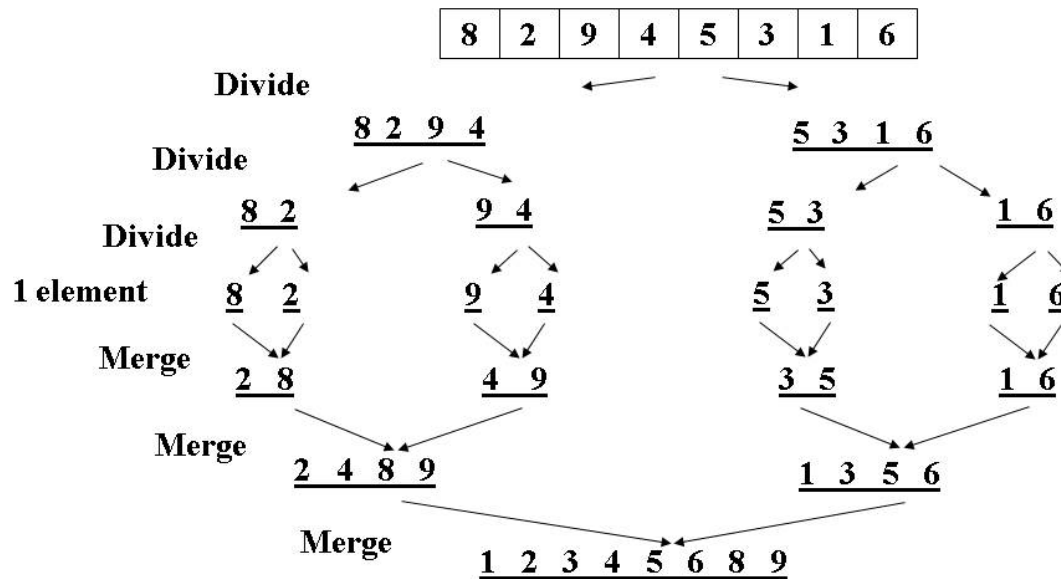
$$T(n) = 2T(n/2) + c_2n + c_3$$

Or more intuitively...

This recurrence comes up often enough you should 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

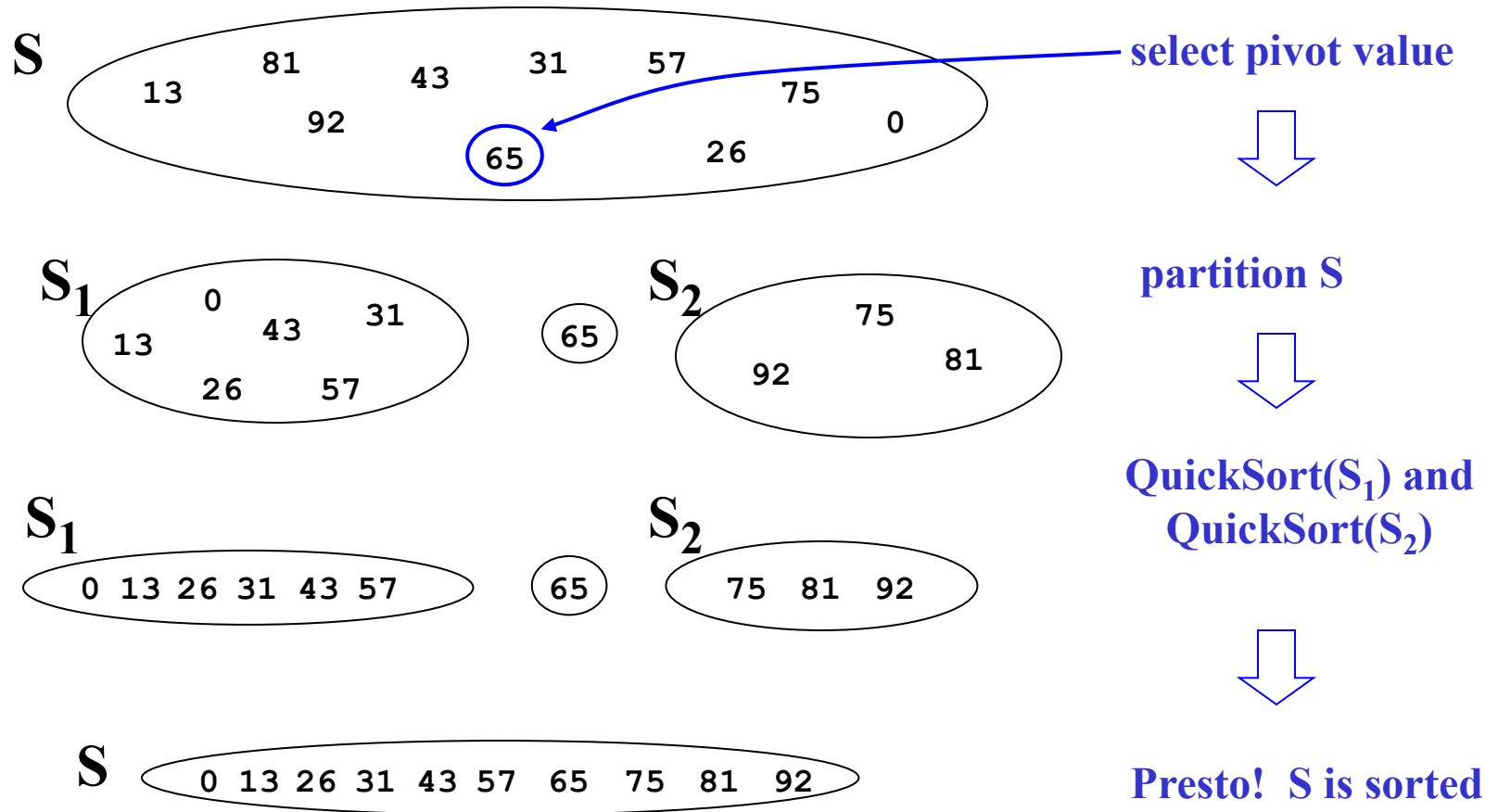
- Also uses divide-and-conquer
 - Recursively chop into halves
 - But, instead of doing all the work as we merge together, we'll do all the work as we recursively split into halves
 - Also unlike MergeSort, does not need auxiliary space
- $O(n \log n)$ on average 😊, but $O(n^2)$ worst-case 😞
 - MergeSort is always $O(n \log n)$
 - So why use QuickSort?
- Can be faster than mergesort
 - Often believed 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
 - Hopefully an element \sim median
 - Good QuickSort performance depends on good choice of pivot; we'll see why later, and talk about good pivot selection later
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”

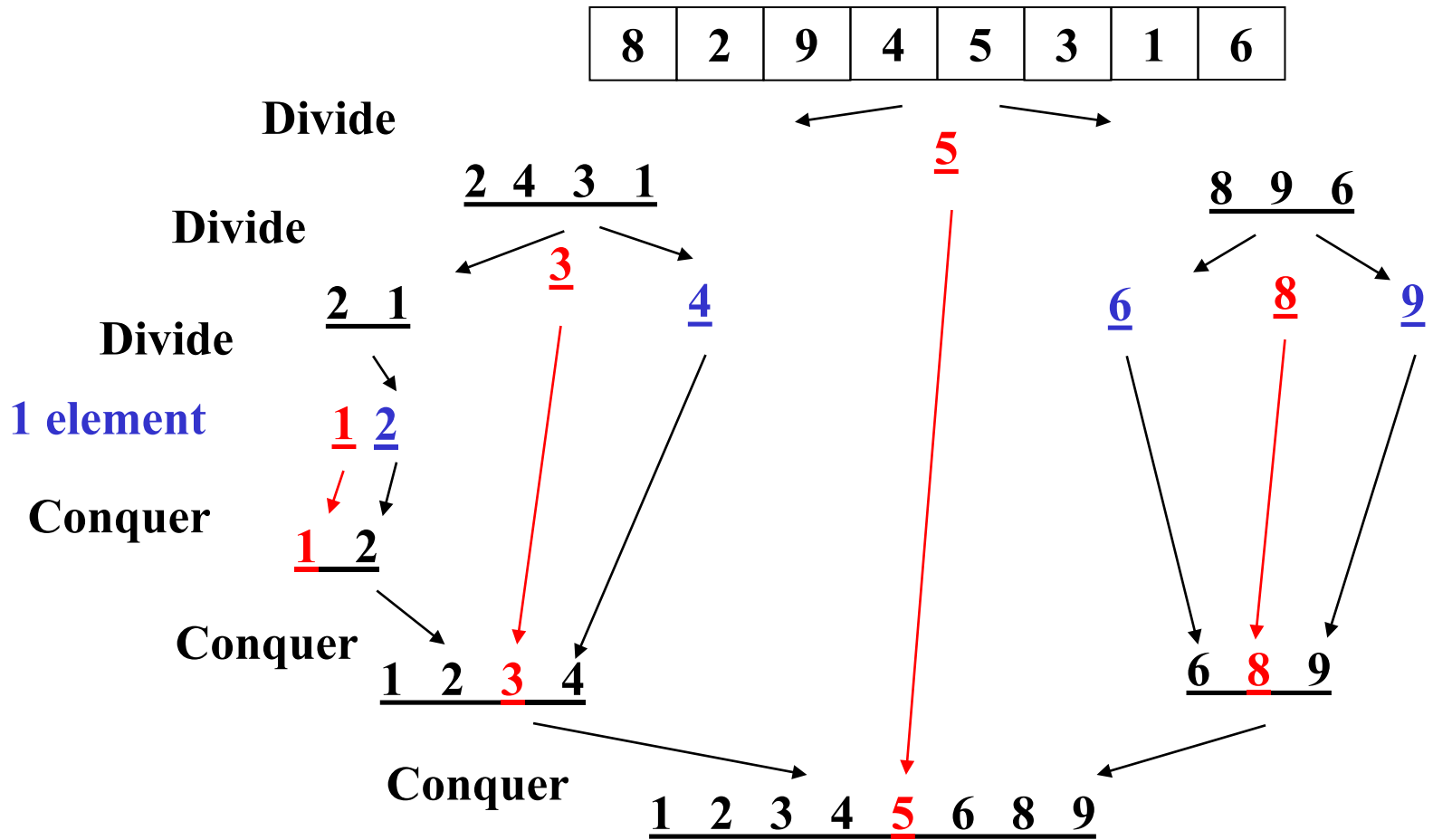
(Alas, there are some details lurking in this algorithm)

Quicksort: Think in terms of sets

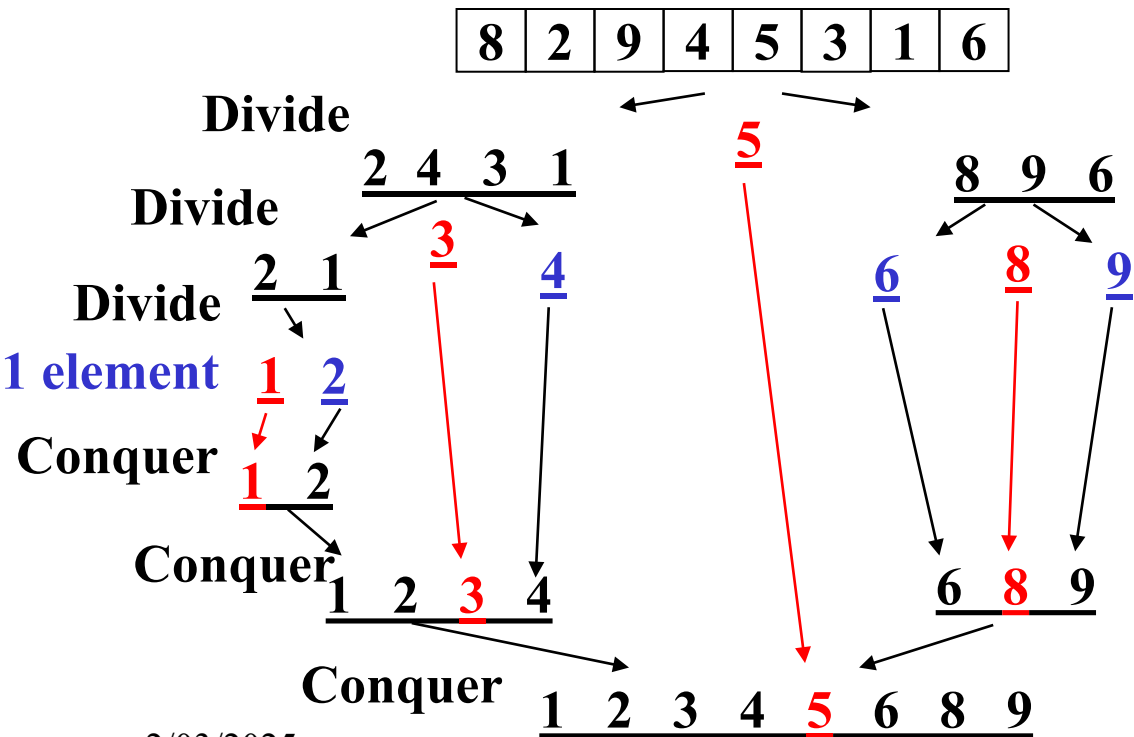
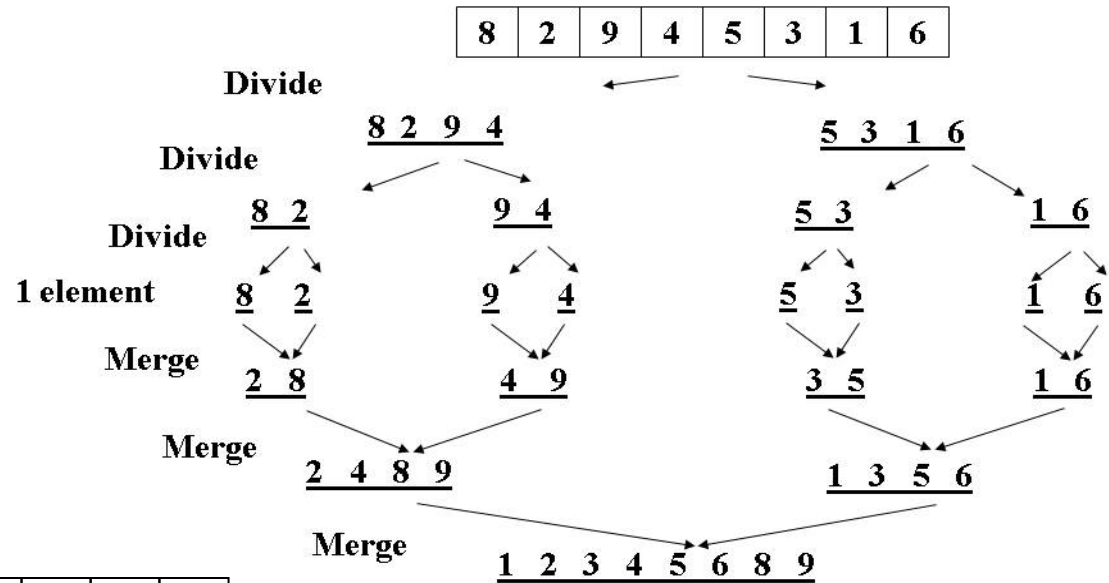


[Weiss]

Quicksort Example, showing recursion



MergeSort Recursion Tree



QuickSort Recursion Tree

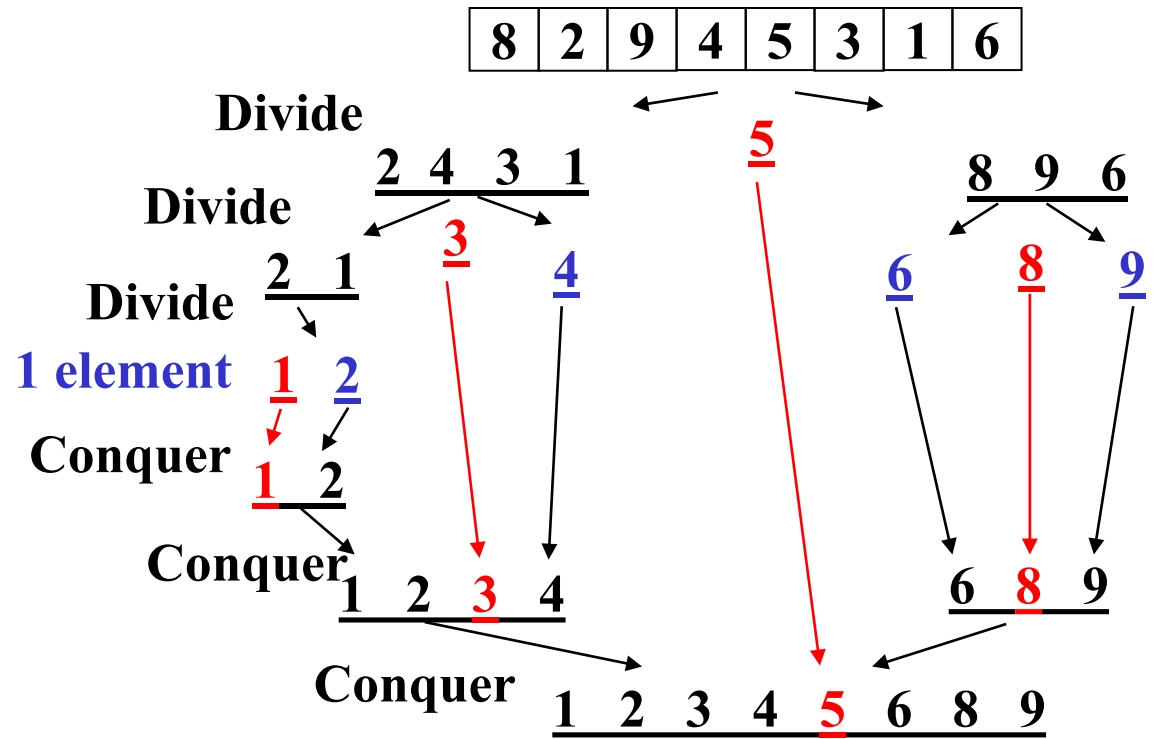
Quicksort Details

We 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

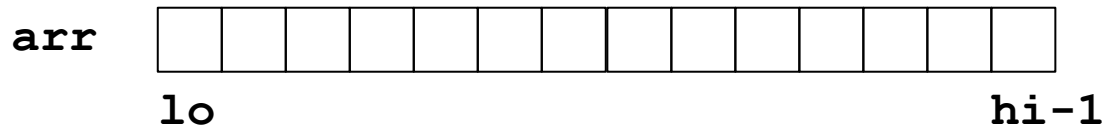
- Best pivot?
 - Median
 - Halve each time
- Worst pivot?
 - Greatest/least element
 - Reduce to problem of size 1 smaller
 - $O(n^2)$



Quicksort: Potential pivot rules

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

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



- Pick `arr[lo]` or `arr[hi-1]`
 - Fast, but worst-case is (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

- That is, given 8, 4, 2, 9, 3, 5, 7 and pivot 5
 - Dividing into left half & right half (based on pivot)
- Conceptually simple, but hardest part to code up correctly
 - After picking pivot, need to partition
 - Ideally in linear time
 - Ideally in place
- Ideas?

Partitioning

- One approach (there are slightly fancier ones):
 1. Swap pivot with `arr[lo]`; move it 'out of the way'
 2. Use two fingers `i` and `j`, starting at `lo+1` and `hi-1` (start & end of range, apart from pivot)
 3. Move from right until we hit something less than the pivot;
belongs on left side
Move from left until we hit something greater than the pivot;
belongs on right side
Swap these two; keep moving inward

```
while (i < j)
    if (arr[j] > pivot) j--
    else if (arr[i] <= pivot) i++
    else swap arr[i] with arr[j]
```
 4. Put pivot back in middle (Swap with `arr[i]`)

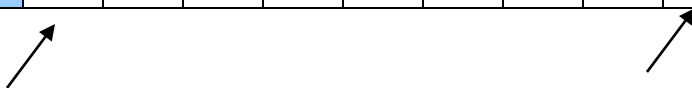
Quicksort Example

- 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

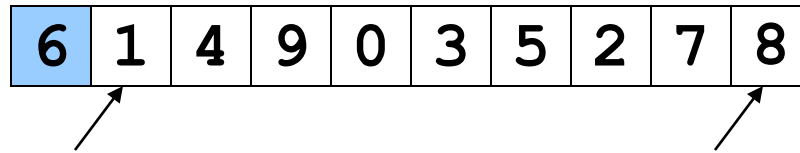
0	1	2	3	4	5	6	7	8	9
6	1	4	9	0	3	5	2	7	8



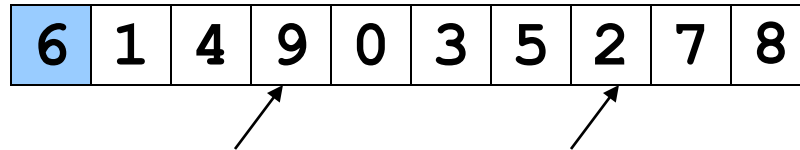
Quicksort Example

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

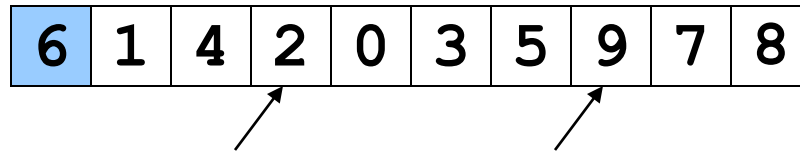
Now partition in place



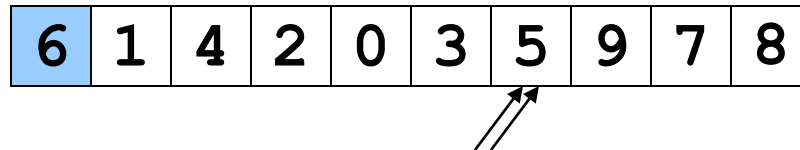
Move fingers



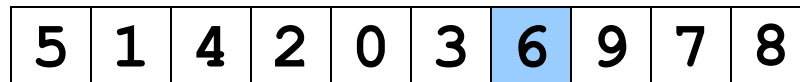
Swap



Move fingers



Move pivot



Quicksort Analysis

- Best-case?
- Worst-case?
- Average-case?

Quicksort Cutoffs

- For small n , all that recursion tends to cost more than doing a quadratic sort
 - Remember asymptotic complexity is for large n
 - Also, recursive calls add a lot of overhead for small n
- Common engineering technique: switch to a different algorithm for subproblems 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

Quicksort Cutoff skeleton

```
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