# CSE 332
# Data Structures & Parallelism

## Comparison Sorting

*Melissa Winstanley*
*Spring 2024*

# 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

# Variations on the basic problem

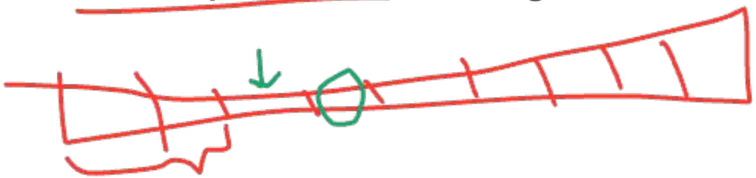
1. Maybe elements are in a linked list (could convert to array and back in linear time, but some algorithms needn't do so)
2. Maybe in the case of ties we should preserve the original ordering
    - Sorts that do this naturally are called stable sorts
    - One way to sort twice, Ex: Sort movies by year, 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: $O(n^2)$ | Fancier algorithms: $O(n \log n)$ | Comparison lower bound: $\Omega(n \log n)$ | Specialized algorithms: $O(n)$ | Handling huge data sets |
|---|---|---|---|---|
| Insertion sort<br>Selection sort<br>... | Heap sort<br>Merge sort<br>Quick sort (avg)<br>... | | Bucket sort<br>Radix sort | 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
    - Now insert 3rd element in order
    - Now insert 4th element in order
    - …
- "Loop invariant": when loop index is **i**, first **i** elements are sorted relative to each other
- Time?

  Best-case $O(n)$ Worst-case $O(n^2)$ "Average" case $O(n^2)$

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

- "Loop invariant": when loop index is **i**, first **i** elements are the **i** smallest elements in sorted order

- Time?

  Best-case $O(n^2)$ Worst-case $O(n^2)$ "Average" case $O(n^2)$

# 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

# Sorting: The Big Picture

| Simple algorithms: $O(n^2)$ | Fancier algorithms: $O(n \log n)$ | Comparison lower bound: $\Omega(n \log n)$ | Specialized algorithms: $O(n)$ | Handling huge data sets |
|---|---|---|---|---|
| Insertion sort<br>Selection sort<br>... | Heap sort<br>Merge sort<br>Quick sort (avg)<br>... | | Bucket sort<br>Radix sort | External sorting |

# Heap sort

- Sorting with a heap is easy!
  - `insert` each `arr[i]`, better yet use **`buildHeap`** O(n)
  - `for(i=0; i < arr.length; i++)` n
    `arr[i] = deleteMin();` log n
- Worst-case running time:

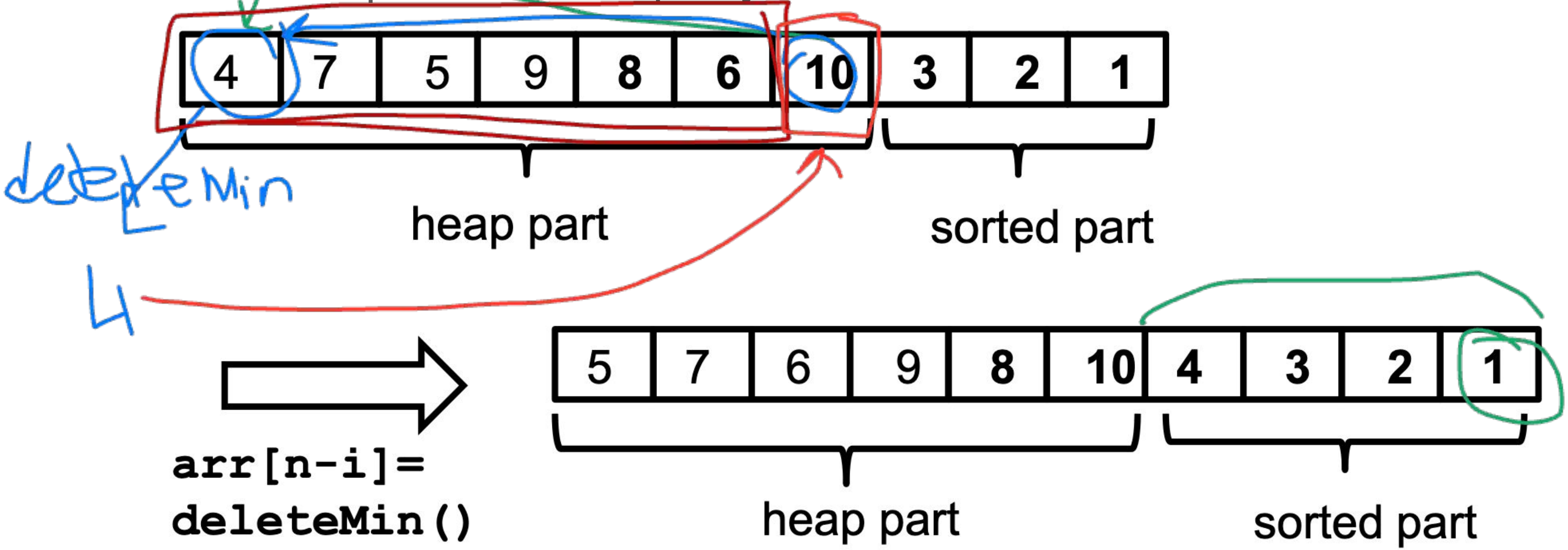
  O(n log n)

- 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!



# Sorting: The Big Picture

| Simple algorithms: $O(n^2)$ | Fancier algorithms: $O(n \log n)$ | Comparison lower bound: $\Omega(n \log n)$ | Specialized algorithms: $O(n)$ | Handling huge data sets |
|---|---|---|---|---|
| Insertion sort<br>Selection sort<br>... | Heap sort<br>Merge sort<br>Quick sort (avg)<br>... | | Bucket sort<br>Radix sort | External sorting |

# 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. **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*]

2. **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

- Uses divide-and-conquer
  - Recursively chop into ~~halves~~ pieces
  - 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(nlogn)
  - 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!

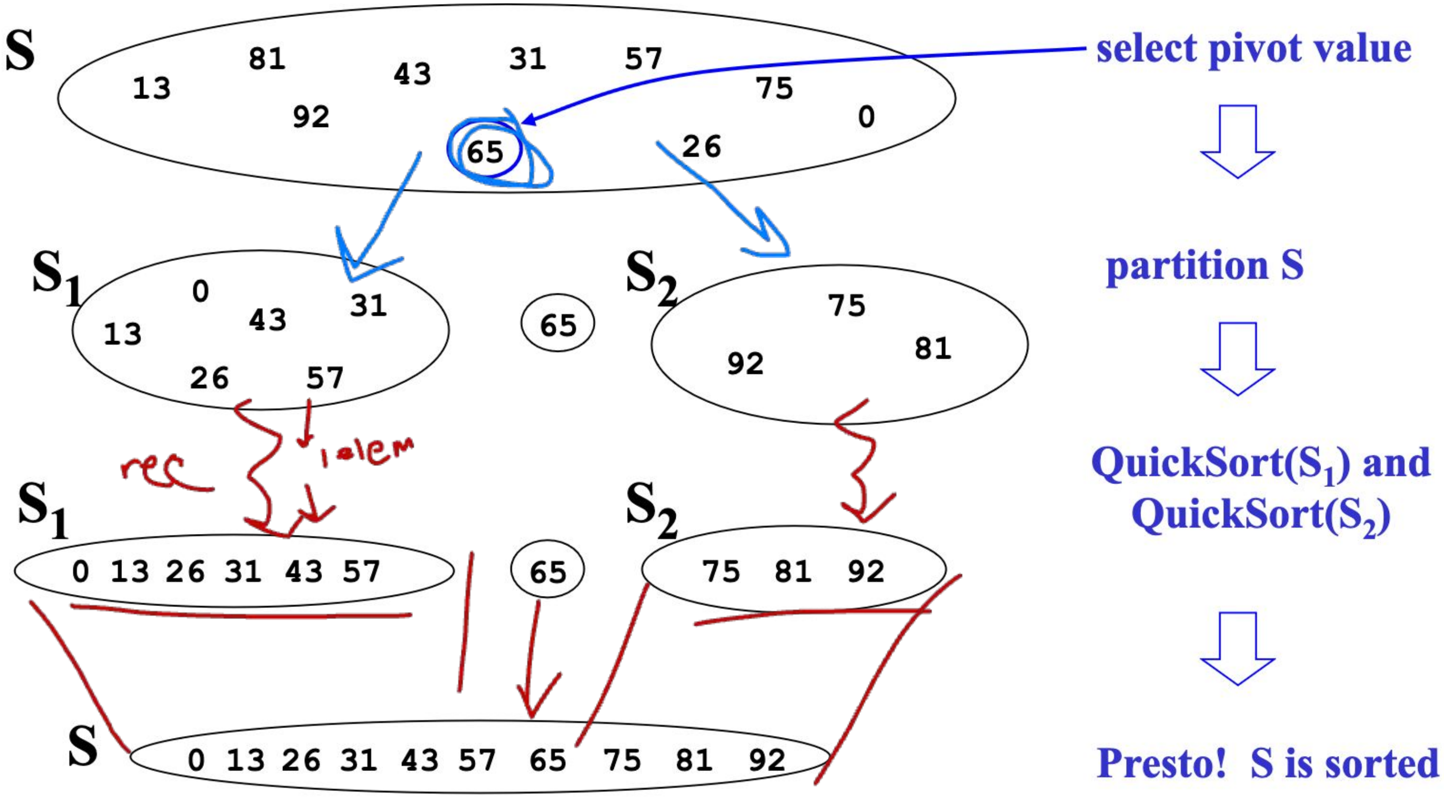
# Quicksort Overview

~~min~~

1. Pick a pivot element
   - Hopefully an element ~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"

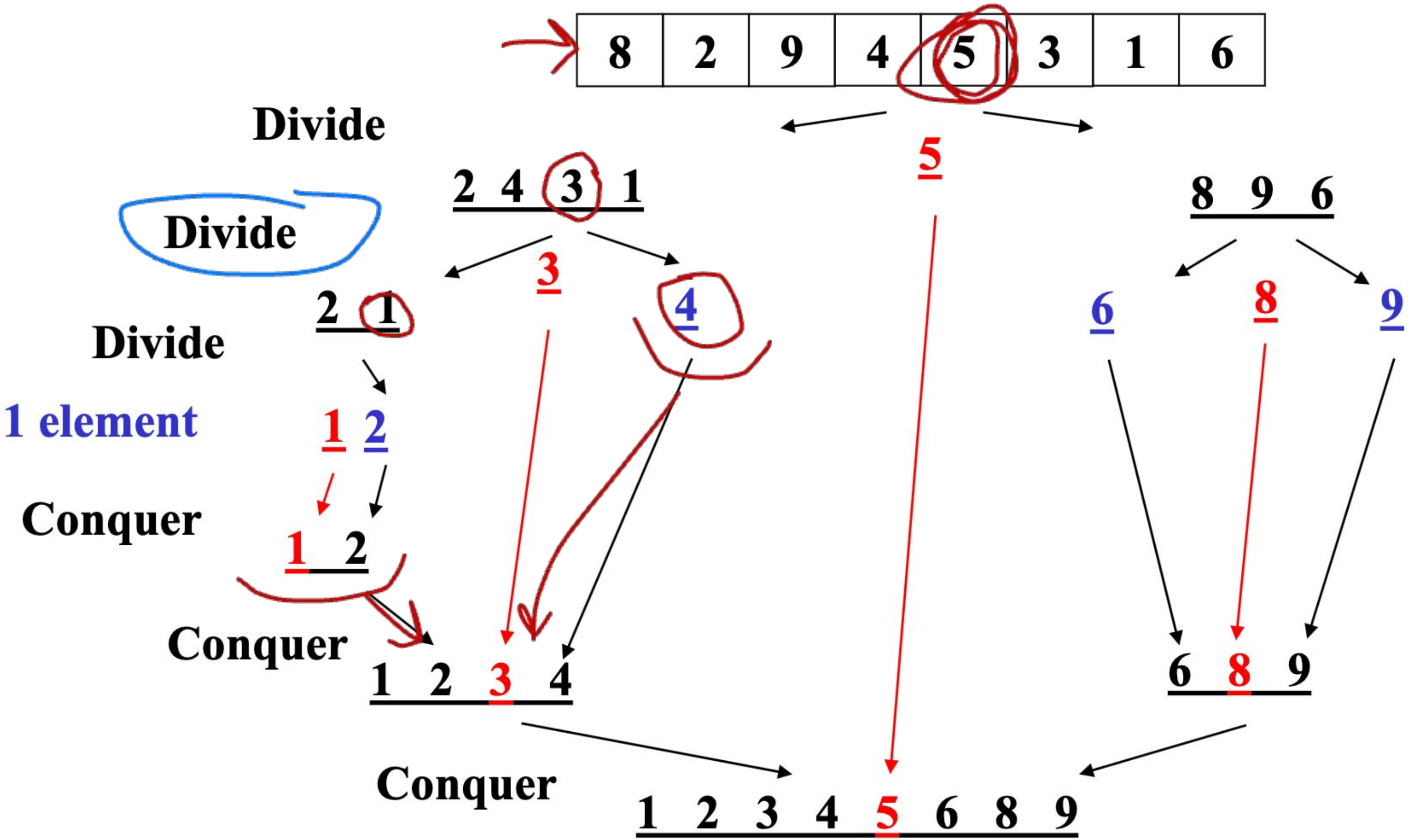
(Alas, there are some details lurking in this algorithm)

# Quicksort: Think in terms of sets




[Weiss]

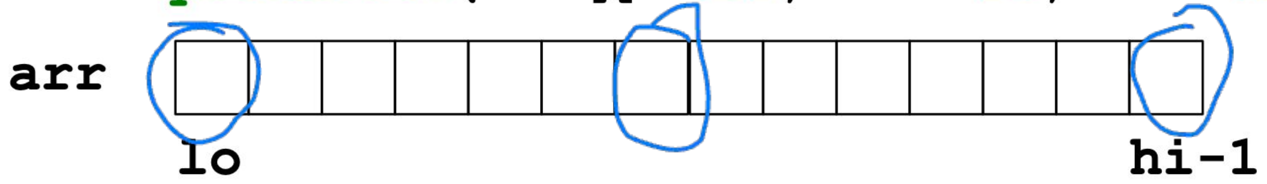
# Quicksort Example, showing recursion



# 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 wel

# 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

1. 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 |

2. 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 | 6 | 1 | 4 | 9 | 0 | 3 | 5 | 2 | 7 | 8 |
| Move fingers | 6 | 1 | 4 | 9 | 0 | 3 | 5 | 2 | 7 | 8 |
| Swap | 6 | 1 | 4 | 2 | 0 | 3 | 5 | 9 | 7 | 8 |
| Move fingers | 6 | 1 | 4 | 2 | 0 | 3 | 5 | 9 | 7 | 8 |
| Move pivot | 5 | 1 | 4 | 2 | 0 | 3 | 6 | 9 | 7 | 8 |

# Quicksort Cutoff

10

```
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

Works for other recursive or parallel algorithms too!

# Sorting: The Big Picture

**Simple algorithms: $O(n^2)$**

Insertion sort
Selection sort
...

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

Heap sort
**Merge sort**
Quick sort (avg)
...

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

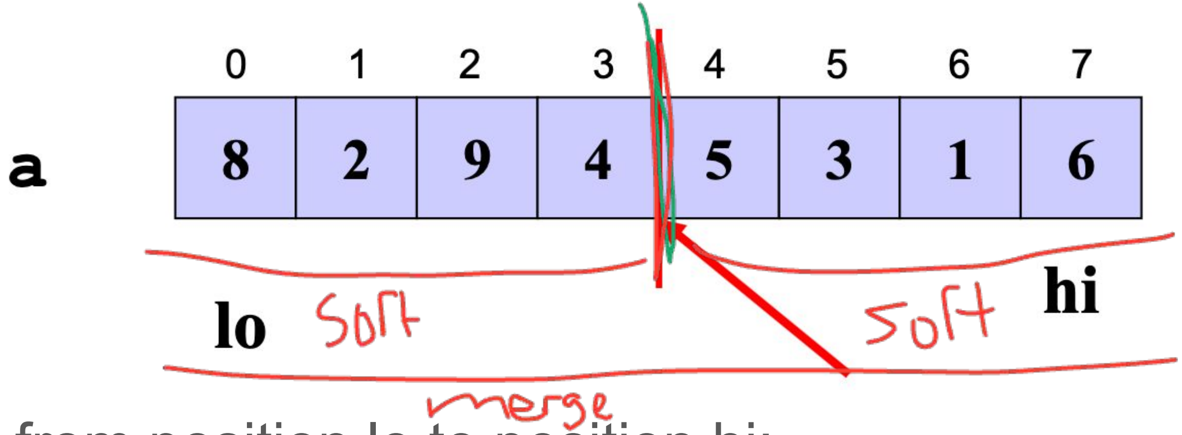
**Specialized algorithms: $O(n)$**

Bucket sort
Radix sort

**Handling huge data sets**

External sorting

# Mergesort



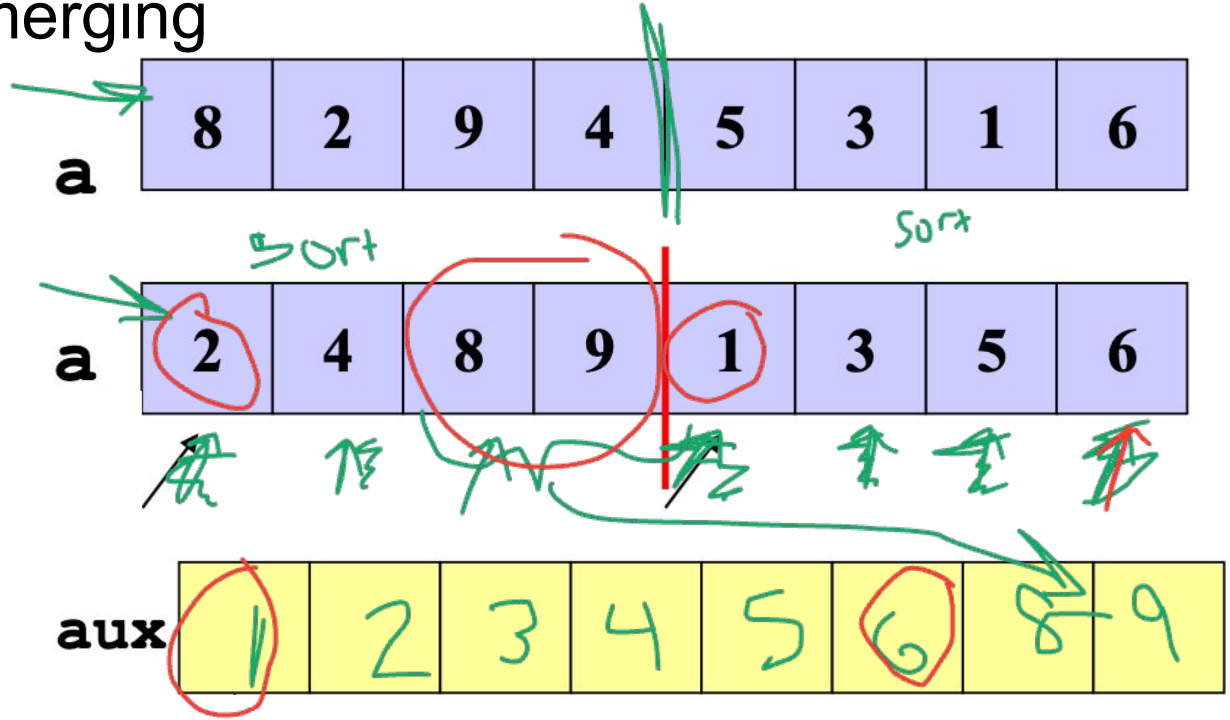

- To sort array from position lo to position hi:
  - If range is 1 element long, it's sorted! (Base case)
  - Else, split into two halves:
    - Sort from lo to (hi+lo)/2
    - Sort from (hi+lo)/2 to hi
    - Merge the two halves together
- Merging takes two sorted parts and sorts everything
  - O(n)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

# Example, focus on merging

Start with:

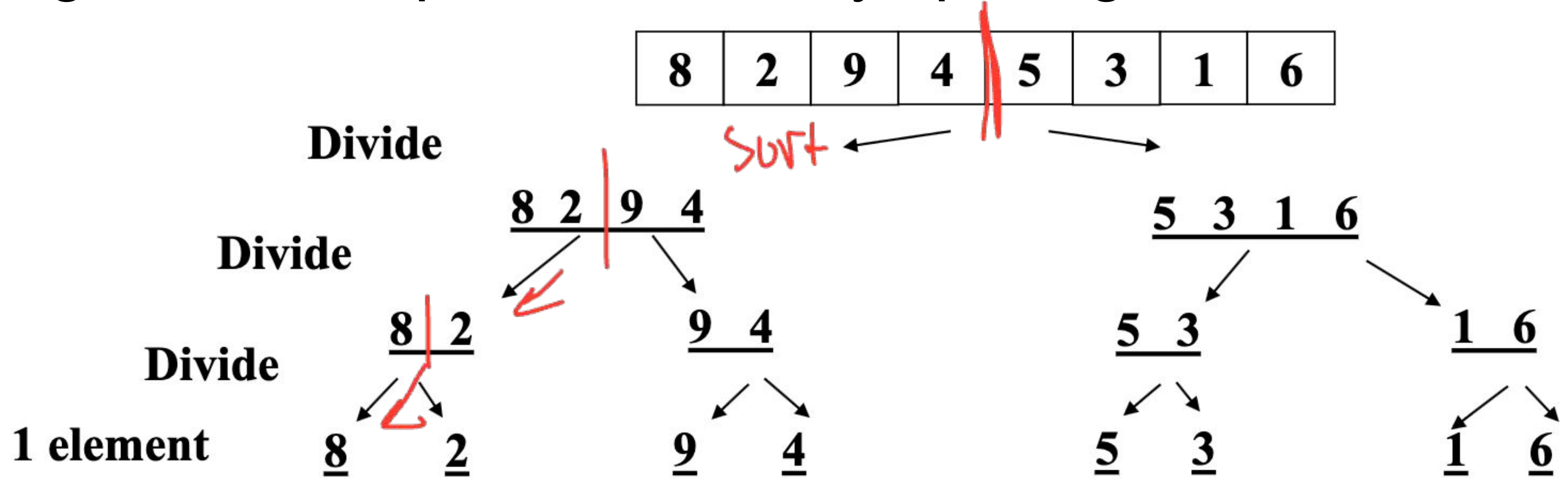
After we return from left and right recursive calls (pretend it works for now)

Merge:
Use 3 “fingers” and 1 more array



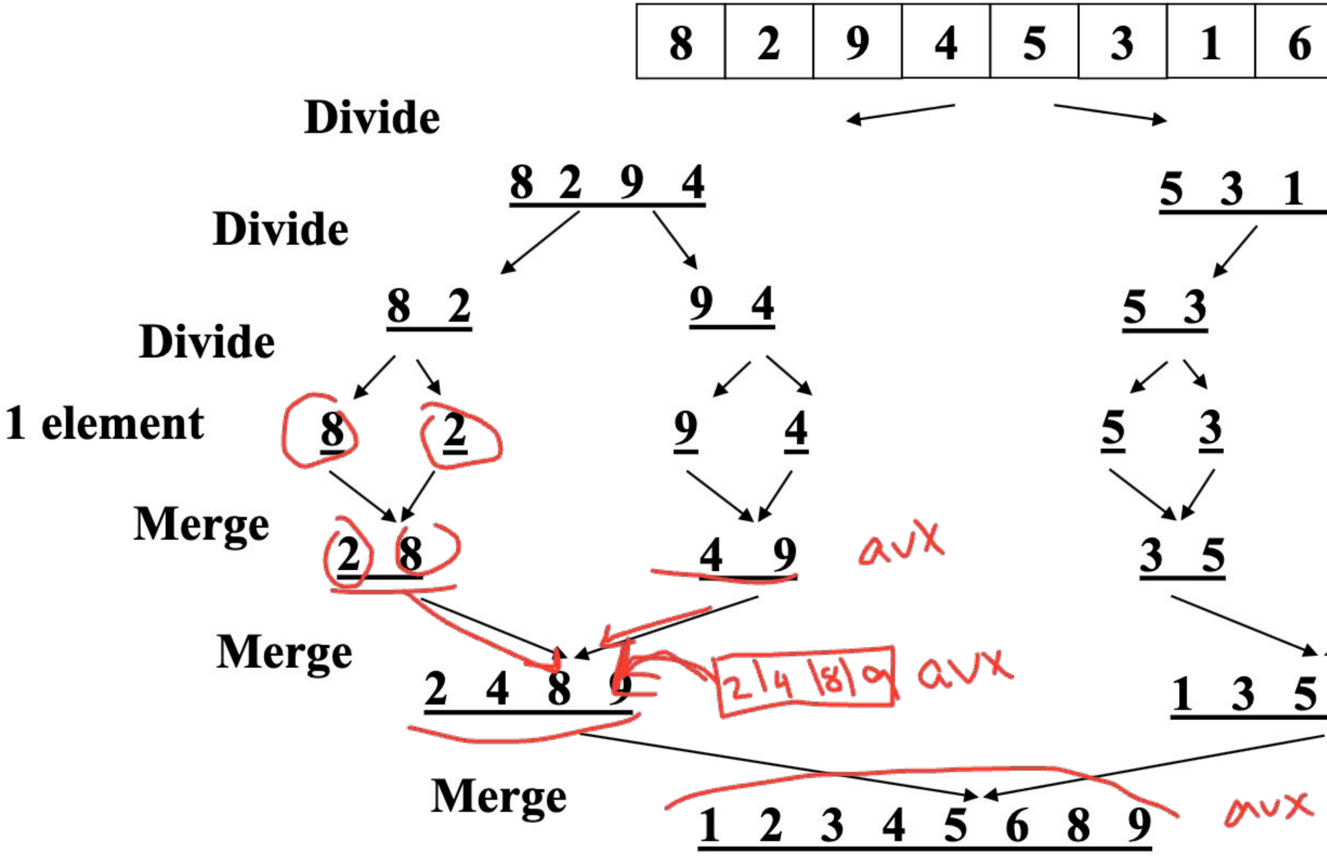

(After merge, copy back to original array)

# Mergesort example: Recursively splitting in half



# Mergesort example: Merge as we return

Don’t forget we need an auxiliary array at each step



# Improving constant factors

- Don’t create a new auxiliary array at each recursive call
  - Reuse the 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
- 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

# 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 a merge of work $n$

Recurrence relation?

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

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

$= n + n \log n$

$= O(n \log n)$

# Quicksort Recurrence

- Best-case?

$T(n) = 2T(n/2) + n$ $\qquad O(n \log n)$

- Worst-case?

$T(n) = T(n-1) + n$ $\qquad O(n^2)$

- Average-case?

$O(n \log n)$

stable

in place