Algorithms & Data Structures

Lesson 10: Comparison Sorting

Marc Gaetano

Edition 2017-2018

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 things" in some order
  - Humans can sort, but computers can sort fast
  - Very common to need data sorted somehow
    - Alphabetical list of people
    - List of countries ordered by population
    - Search engine results by relevance
    - …
- 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

Why Study Sorting in this Class?

- Unlikely you will ever need to re-implement a sorting algorithm yourself
  - Standard libraries will generally implement one or more (Java implements 2)

- You will almost certainly use sorting algorithms
  - Important to understand relative merits and expected performance

- Excellent set of algorithms for practicing analysis and comparing design techniques
  - Classic part of a data structures class, so you'll be expected to know it

The main problem, stated carefully

For now, assume 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
- A comparison function (consistent and total)

Effect:

- Reorganize the elements of **`A`** such that for any **`i`** and **`j`**, if **`i < j`** then **`A[i] ≤ A[j]`**
- (Also, **`A`** must have exactly 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 ties need to be resolved by "original array position"
   - Sorts that do this naturally are called stable sorts
   - Others could tag each item with its original position and adjust comparisons accordingly (non-trivial constant factors)

3. Maybe we must not use more than *O*(1) "auxiliary space"
   - Sorts meeting this requirement are called in-place sorts

4. Maybe we can do more with elements than just compare
   - Sometimes leads to faster algorithms

5. Maybe we have too much data to fit in memory
   - Use an "external sorting" algorithm

Sorting: The Big Picture

Surprising amount of neat stuff to say about sorting:

Simple algorithms: $O(n^2)$

- Insertion sort
- Selection sort
- Shell sort
- …

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

- Heap sort
- Merge sort
- Quick sort
- …

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
  - Now insert 3rd element in order
  - Now insert 4th element in order
  - …

- "Loop invariant": when loop index is **`i`**, first **`i`** elements are sorted

- Exercise: running 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, 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
- Exercise: running 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 large arrays that are not already almost sorted*
  - Insertion sort may do well on small arrays

Heap sort

- Sorting with a heap is easy:
  - **`insert`** each **`arr[i]`**, or better yet use **`buildHeap`**
  - ```
    for(i=0; i < arr.length; i++)
        arr[i] = deleteMin();
    ```

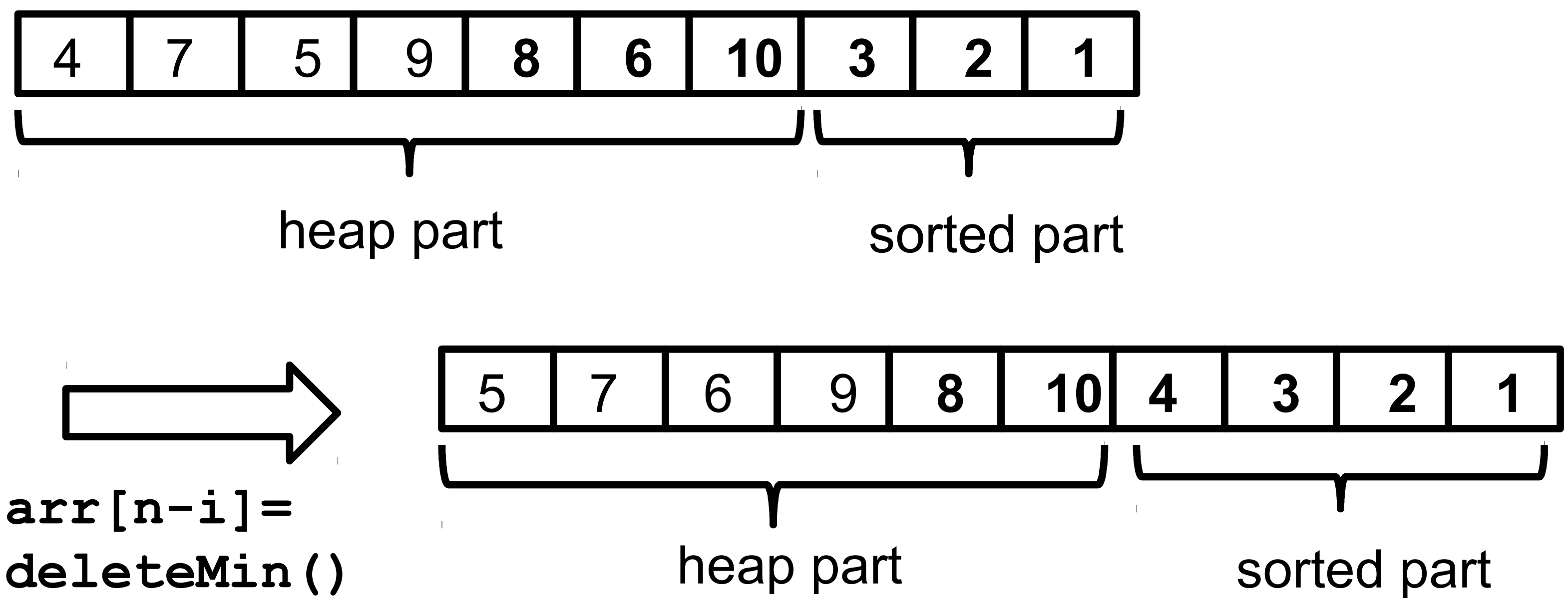
- Worst-case running time: $O(n \log n)$

- 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]`**
  - That array location isn't needed for the heap anymore!



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

(This technique has a *long* history.)

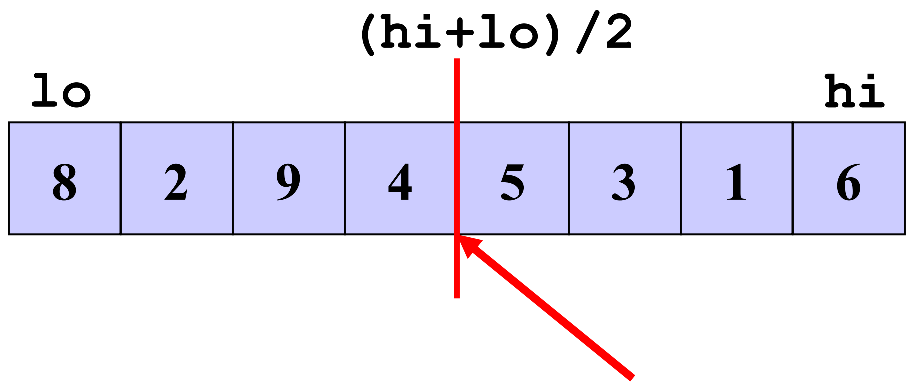
Divide-and-Conquer Sorting

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 less-than pivot
   and greater-than pivot
   Sort the two divisions (recursively on each)
   Final state is:
   sorted-less-than then pivot then sorted-greater-than

Merge sort



- To sort array from position **`lo`** to position **`hi`**:
  - If range is 1 element long, it is already sorted! (Base case)
  - Else:
    - Sort from **`lo`** to **`(hi+lo)/2`**
    - Sort from **`(hi+lo)/2`** to **`hi`**
    - Merge the two halves together

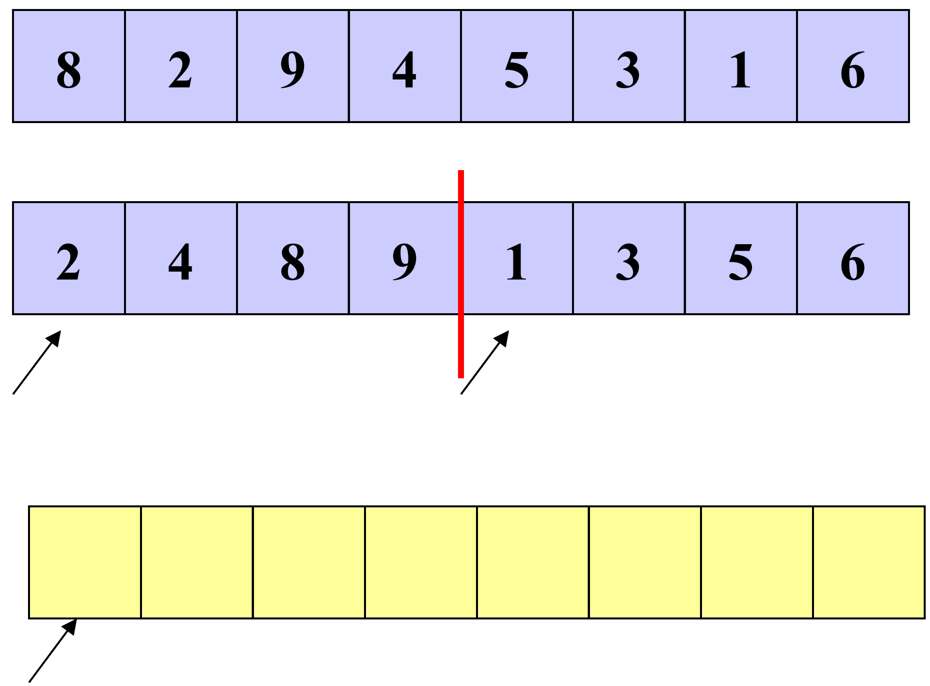
- Merging takes two sorted parts and sorts everything
  - $O(n)$ but requires auxiliary space…

Example, focus on merging

Start with:

After recursion:
(not magic)

Merge:
Use 3 “fingers”
and 1 more array



(After merge, copy back to original array)

Example, focus on merging

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							

Example, focus on merging

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**						

Example, focus on merging

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**					

Example, focus on merging

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

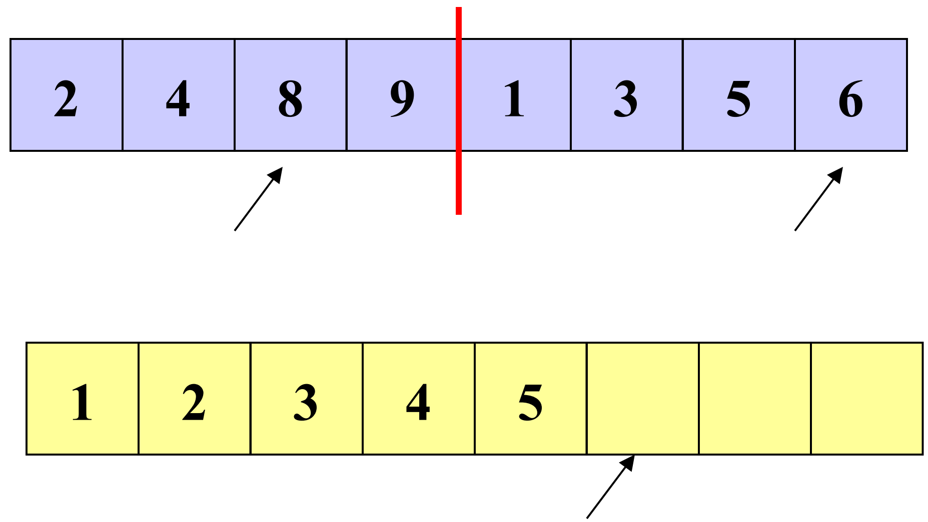
1	2	3	4				

Example, focus on merging

Start with:

After recursion:
(not magic)

Merge:
Use 3 “fingers”
and 1 more array



Example, focus on merging

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**		

Example, focus on merging

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	

Example, focus on merging

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

Example, focus on merging

Start with:

8	2	9	4	5	3	1	6

After recursion:
(not magic)

2	4	8	9	1	3	5	6

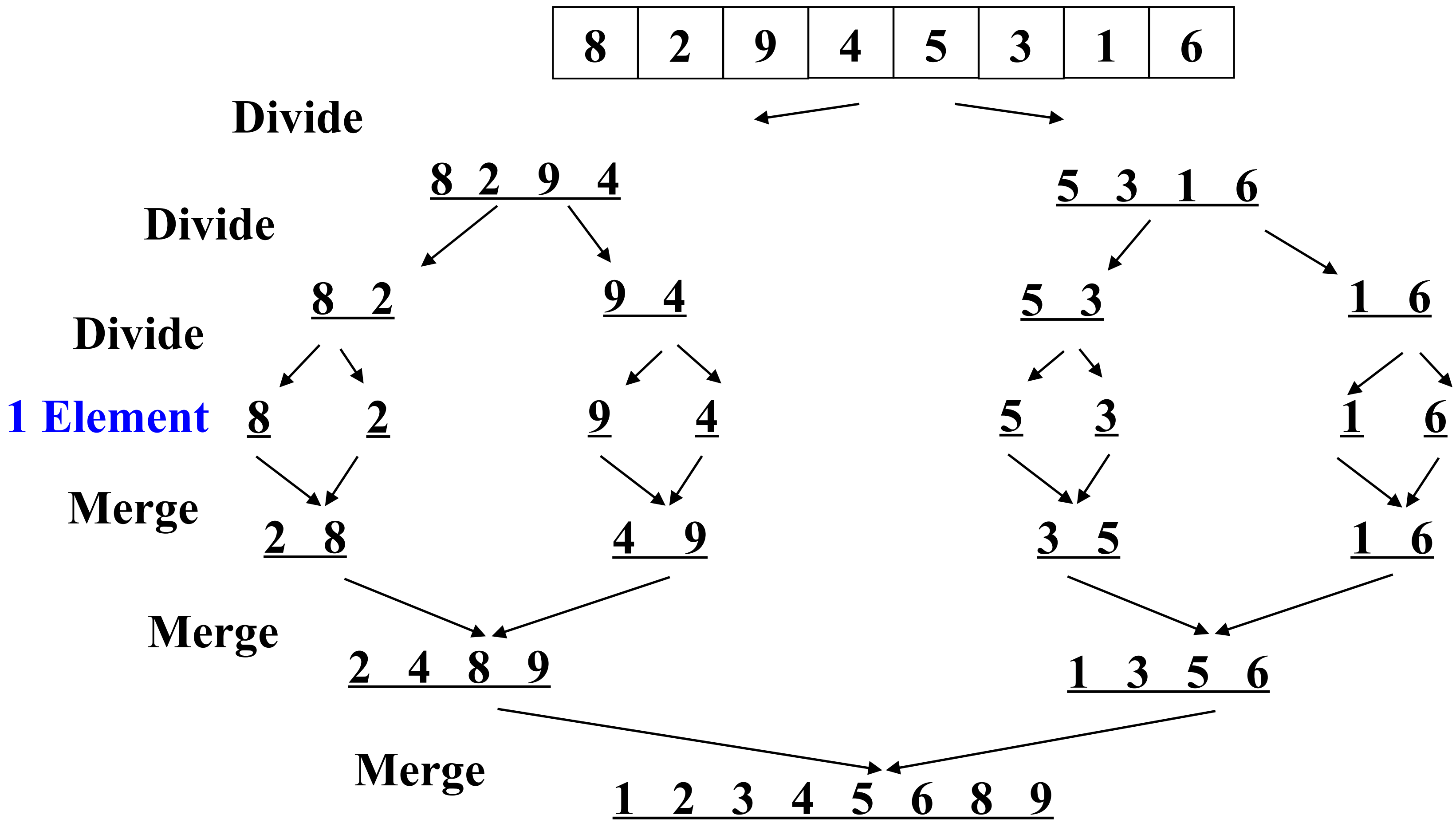
Merge:
Use 3 “fingers”
and 1 more array

1	2	3	4	5	6	8	9

(After merge,
copy back to
original array)

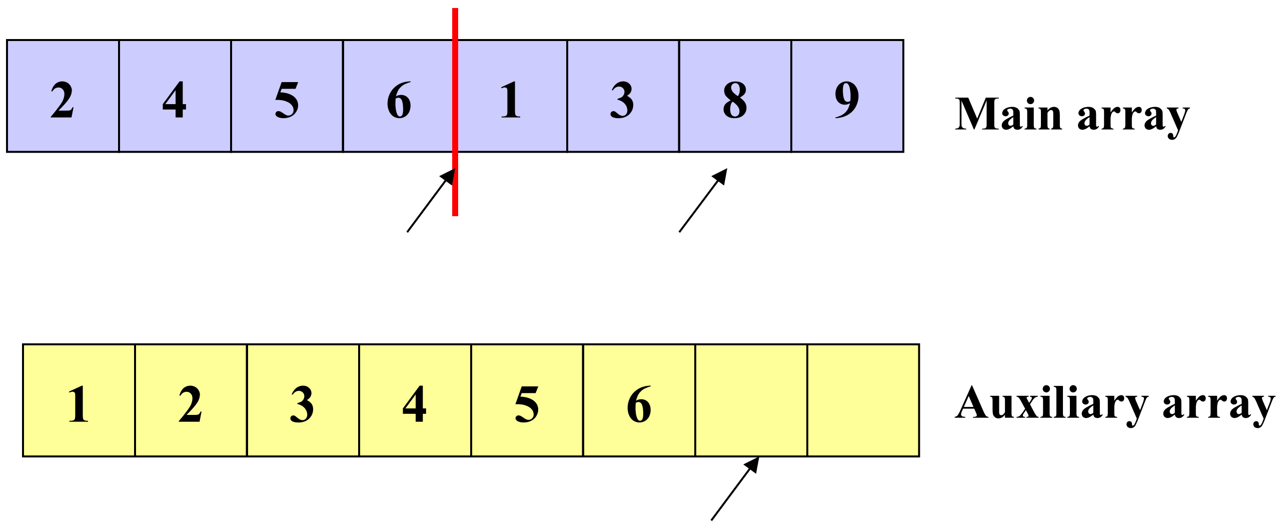
1	2	3	4	5	6	8	9

Example, Showing Recursion



Some details: saving a little time

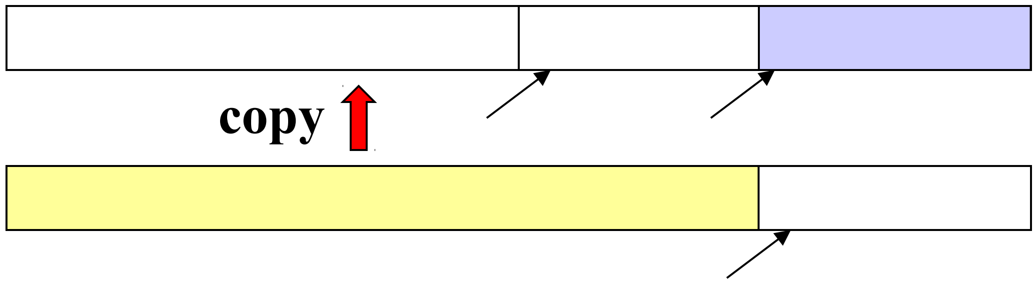
- What if the final steps of our merge looked like this:



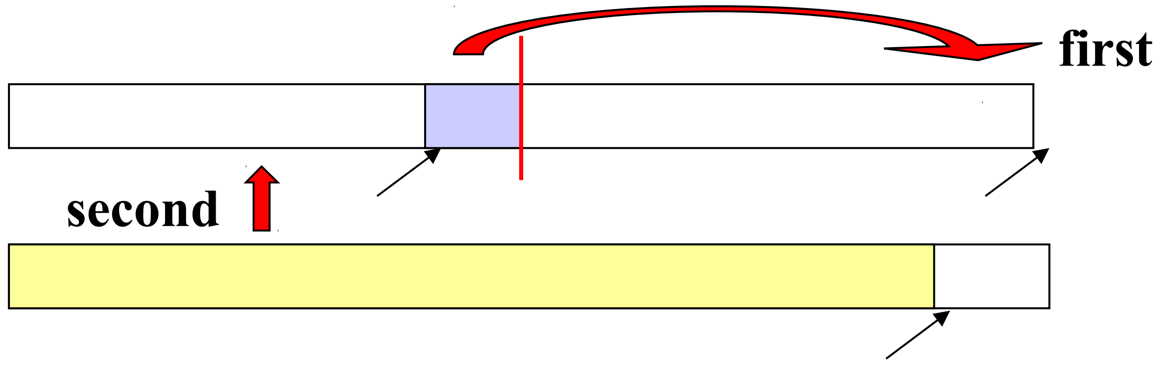
- Wasteful to copy to the auxiliary array just to copy back…

Some details: saving a little time

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



- If right-side finishes first, copy dregs into right then copy back



Some details: Saving Space and Copying

Simplest / Worst:

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

Better:

Use a new auxiliary array of size **`n`** for every merging stage

Better:

Reuse same auxiliary array of size **`n`** for every merging stage

Best (but a little tricky):

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

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

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:

$$T(1) = c_1$$

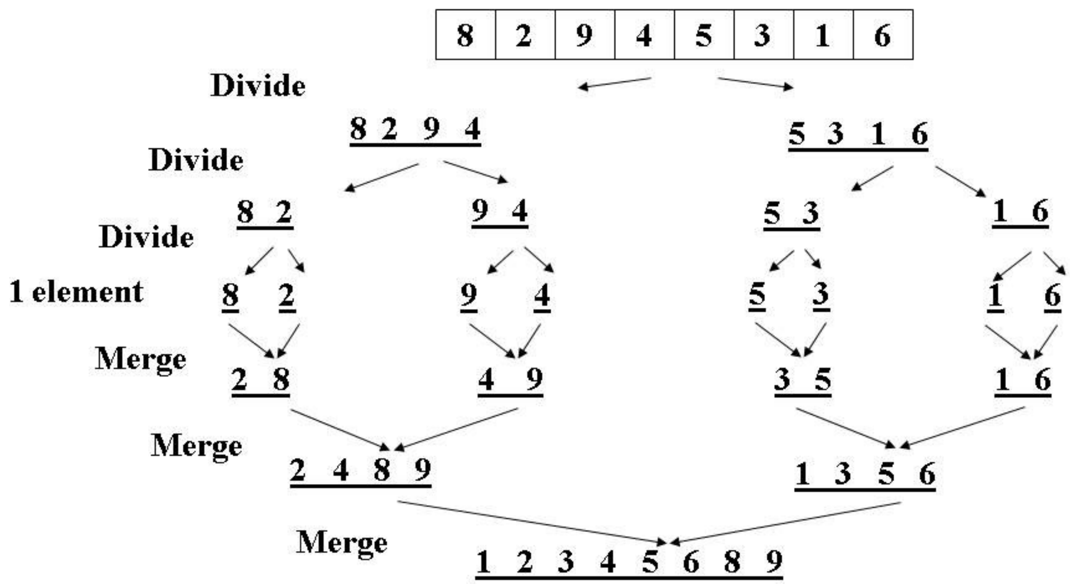
$$T(n) = 2T(n/2) + c_2 n$$

Analysis intuitively

This recurrence is common 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



Analysis more formally

(One of the recurrence classics)

For simplicity let constants be 1 (no effect on asymptotic answer)

$T(1) = 1$

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

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

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

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

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

….

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

So total is $2^k T(n/2^k) + kn$ where

$n/2^k = 1$, i.e., $\log n = k$

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

$= n + n \log n$

$= O(n \log n)$

Quick sort

- A divide-and-conquer algorithm
  - Recursively chop into two pieces
  - Instead of doing all the work as we merge together, we will do all the work as we recursively split into halves
  - Unlike merge sort, does not need auxiliary space

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

- Faster than merge sort in practice?
  - Often believed so
  - Does fewer copies and more comparisons, so it depends on the relative cost of these two operations!

Quicksort Overview

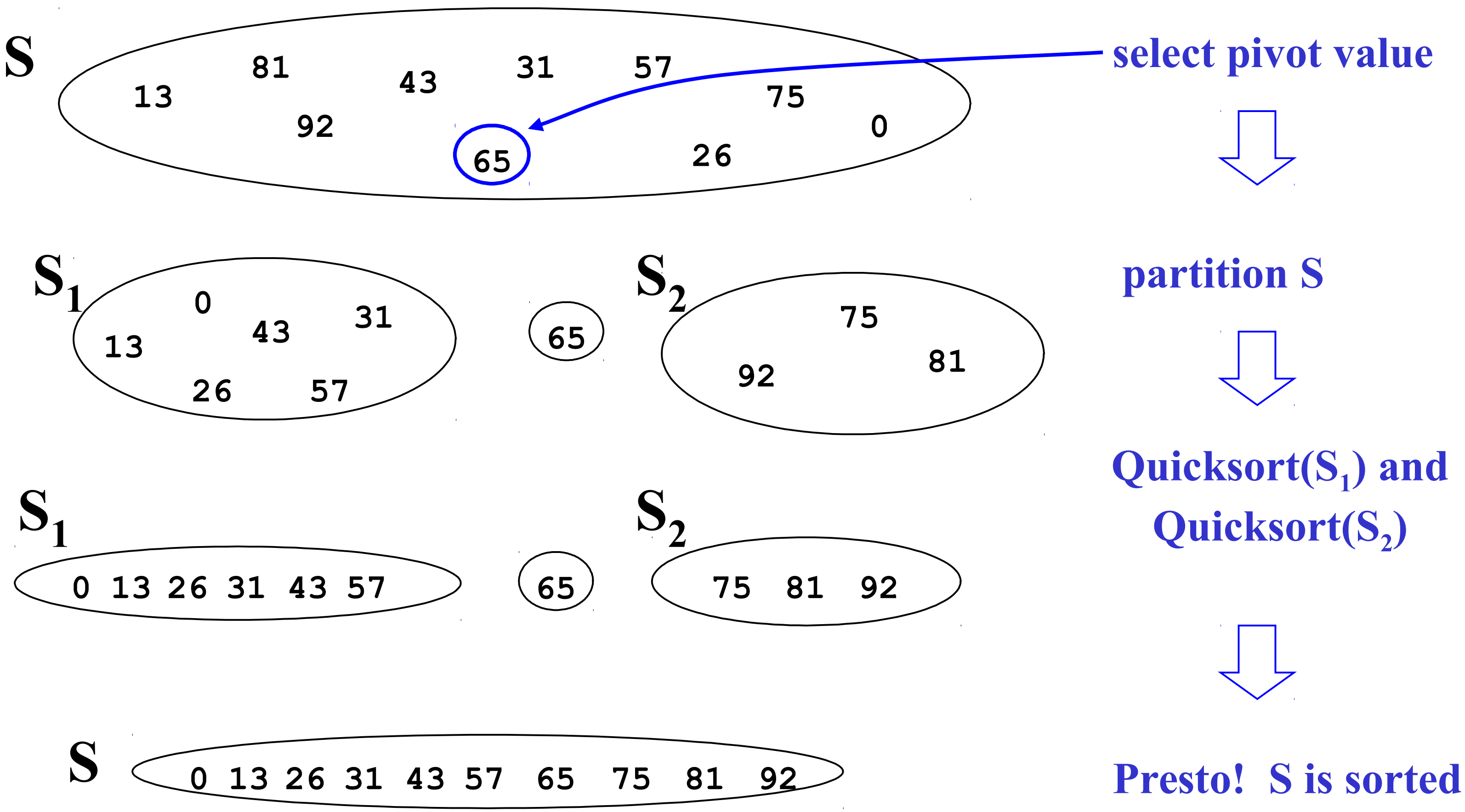
1. Pick a pivot element

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

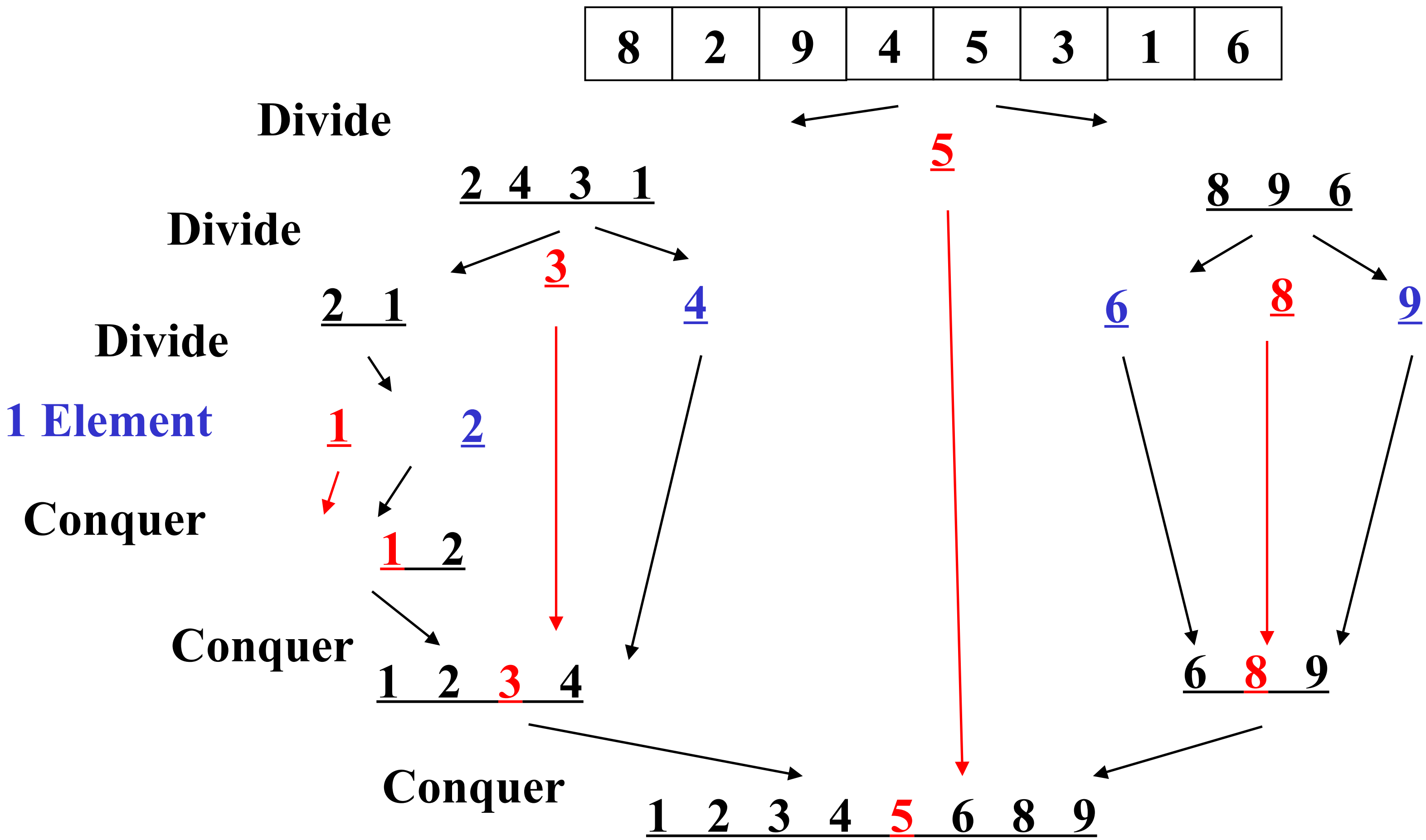
4. The answer is, “as simple as A, B, C”

Think in Terms of Sets



[Weiss]

Example, Showing Recursion



Details

Have not yet explained:

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

- How to implement partitioning
  - In linear time
  - In place

Pivots

- Best pivot?
  - Median
  - Halve each time

8	2	9	4	5	3	1	6

5

2 4 3 1   8 9 6

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

8	2	9	4	5	3	1	6

1

8 2 9 4 5 3 6

Potential pivot rules

While sorting **`arr`** from **`lo`** to **`hi-1`** …

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

Partitioning

- Conceptually simple, but hardest part to code up correctly
  - After picking pivot, need to partition in linear time in place

- One approach (there are slightly fancier ones):
  1. Swap pivot with **`arr[lo]`**
  2. Use two fingers **`i`** and **`j`**, starting at **`lo+1`** and **`hi-1`**
  3. **`while (i < j)`**

```
         if (arr[j] > pivot) j--
         else if (arr[i] < pivot) i++
         else swap arr[i] with arr[j]
```

  4. Swap pivot with **`arr[i]`**

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

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

Analysis

- Best-case: Pivot is always the median

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

  $T(n)=2T(n/2) + n$ -- linear-time partition

  Same recurrence as merge sort: $O(n \log n)$

- Worst-case: Pivot is always smallest or largest element

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

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

  Basically same recurrence as selection sort: $O(n^2)$

- Average-case (e.g., with random pivot)
  - $O(n \log n)$, not responsible for proof (in text)

Cutoffs

- For small *n*, all that recursion tends to cost more than doing a quadratic sort
  - Remember asymptotic complexity is for large *n*

- Common engineering technique: switch algorithm below a cutoff
  - Reasonable rule of thumb: use **insertion sort** for $n < 10$

- Notes:
  - Could also use a cutoff for merge sort
  - Cutoffs are also the norm with parallel algorithms
    - Switch to sequential algorithm
  - None of this affects asymptotic complexity

Cutoff pseudocode

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

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

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

How Fast Can We Sort?

- Heapsort & mergesort have $O(n\ \mathtt{log}\ n)$ worst-case running time
- Quicksort has $O(n\ \mathtt{log}\ n)$ average-case running time
- These bounds are all tight, actually $\Theta(n\ \mathtt{log}\ n)$
- Comparison sorting in general is $\Omega\ (n\ \mathtt{log}\ n)$
  - An amazing computer-science result: proves all the clever programming in the world cannot comparison-sort in linear time

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 Selection sort Shell sort …	Heap sort Merge sort Quick sort (avg) …		Bucket sort Radix sort	External sorting

How???

- Change the model – assume more than "compare(a,b)"

Conclusion on Sorting

- Simple $O(n^2)$ sorts can be fastest for small n
  - Selection sort, Insertion sort (latter linear for mostly-sorted)
  - Good for "below a cut-off" to help divide-and-conquer sorts
- $O(n$ **`log`** $n)$ sorts
  - Heap sort, in-place but not stable nor parallelizable
  - Merge sort, not in place but stable and works as external sort
  - Quick sort, in place but not stable and $O(n^2)$ in worst-case
    - Often fastest, but depends on costs of comparisons/copies
- Ω $(n$ **`log`** $n)$ is worst-case and average lower-bound for sorting by comparisons
- Non-comparison sorts
  - Bucket sort good for small number of possible key values
  - Radix sort uses fewer buckets and more phases
- Best way to sort? It depends!