CSE332: Data Abstractions

Lecture 13: Comparison Sorting

Tyler Robison

Summer 2010

The Big Picture

Quite a bit to cover

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

We'll start with simple sorts

Selection sort

- Idea: At the **`k`**th step, 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? **Recurrence always: T(1) = O(1) and T(n) = O(n) + T(n-1)**

Best-case $\underline{\mathbf{O(n^2)}}$ Worst-case $\underline{\mathbf{O(n^2)}}$ "Average" case $\underline{\mathbf{O(n^2)}}$

Insertion Sort

- Idea: At the **`k`**th step put the **`k`**th element in the correct place 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
- Time?

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

Starts sorted Starts reverse sorted

Mystery

This is one implementation of which sorting algorithm (for ints)?

```
void mystery(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;
  }
}
```

Note: Like with heaps, “moving the hole” is faster than unnecessary swapping (constant factor)

Insertion vs. Selection

- They are different algorithms; different ideas
- They solve the same problem
- They 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 larger arrays that are not already almost sorted*
  - Small arrays may do well with Insertion sort

Aside: Why we're not going to cover Bubble Sort

- Not really what a "normal person" would think of
- 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
- So people seem to teach it just because someone taught it to them

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

A Fancier Sort: Heap sort

- As you saw on project 2, sorting with a heap isn't too bad:
  - `insert` each `arr[i]`, better yet `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 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!
  - We know the heap won't grow back to that size

4	7	5	9	**8**	**6**	**10**	**3**	**2**	**1**

heap part sorted part

```
arr[n-i]=
deleteMin()
```

5	7	6	9	**8**	**10**	**4**	**3**	**2**	**1**

heap part sorted part

"AVL sort"

- We could also use a balanced tree to:
  - **`Insert`** each element: total time $O(n \log n)$
  - Repeatedly **`deleteMin`**: total time $O(n \log n)$

- But this cannot be made in-place and has worse constant factors than heap sort
  - Heap sort is better
  - Both are $O(n \log n)$ in worst, best, and average case
  - Neither parallelizes well

- How about sorting with a hash table?

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 each half, split into halves…

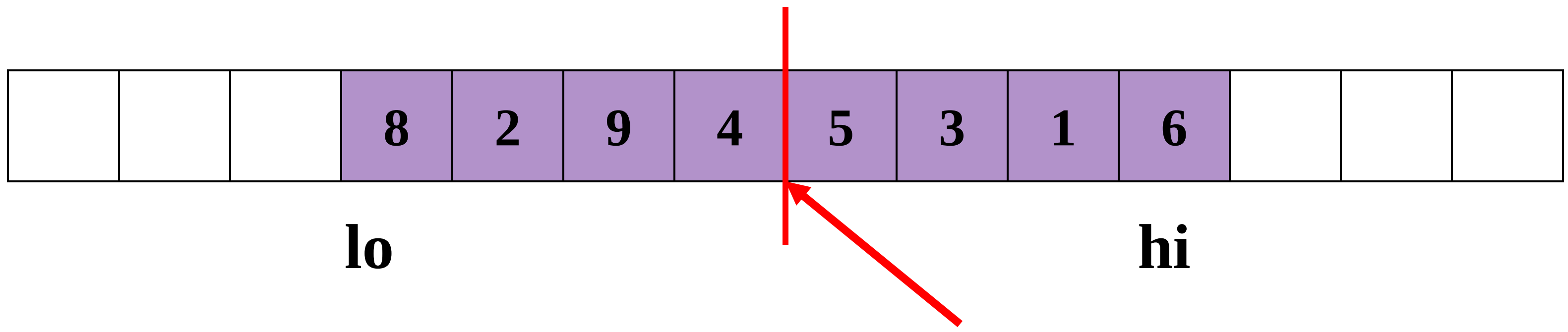
Other fancy sorts: Divide-and-conquer sorting

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)
   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 2 halves:
    - Call Mergesort on left half; when it returns, that half is sorted
    - Call Mergesort on right half; when it returns, that half is sorted
    - Merge the two halves together
- The Merge step takes two sorted parts and sorts everything together
  - $O(n)$ (per merge) but requires auxiliary space…

The Merging part

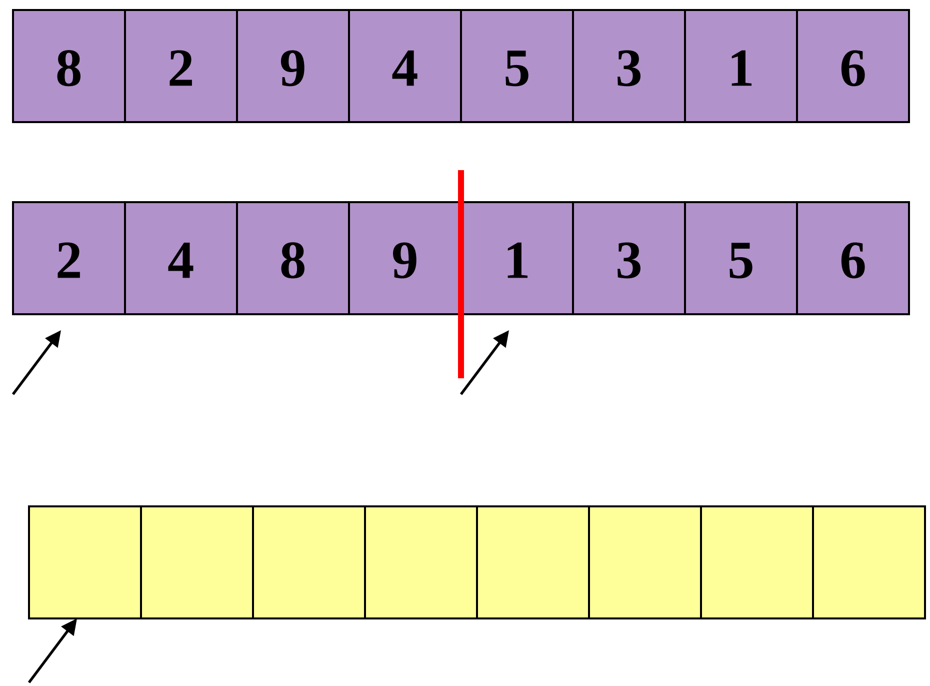
Start with:

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

Merge:

Use 3 “fingers”

and 1 more array



The Merging part

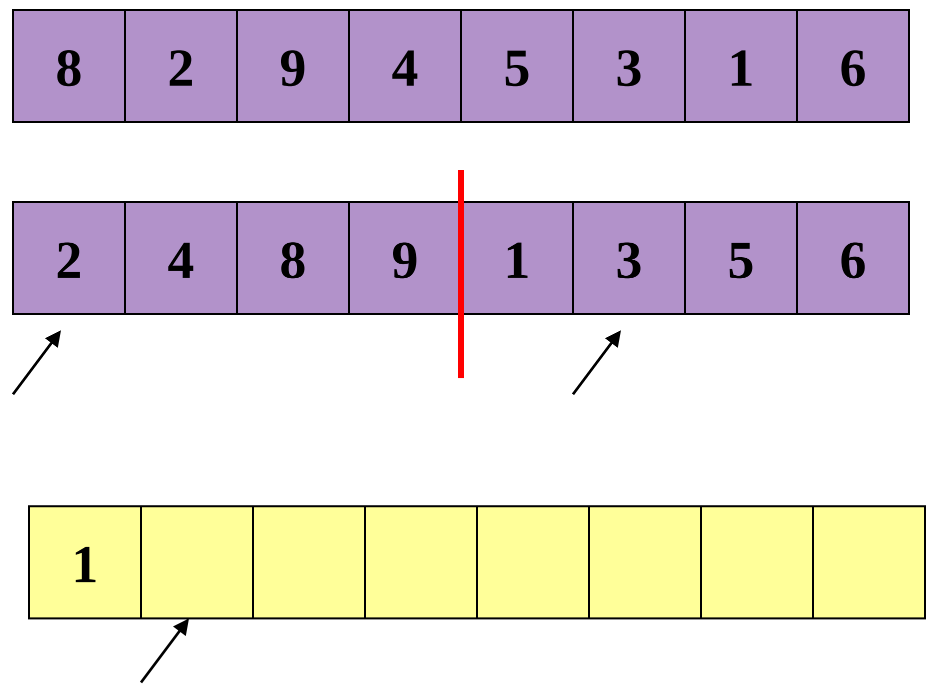
Start with:

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

Merge:

Use 3 “fingers”

and 1 more array



The Merging part

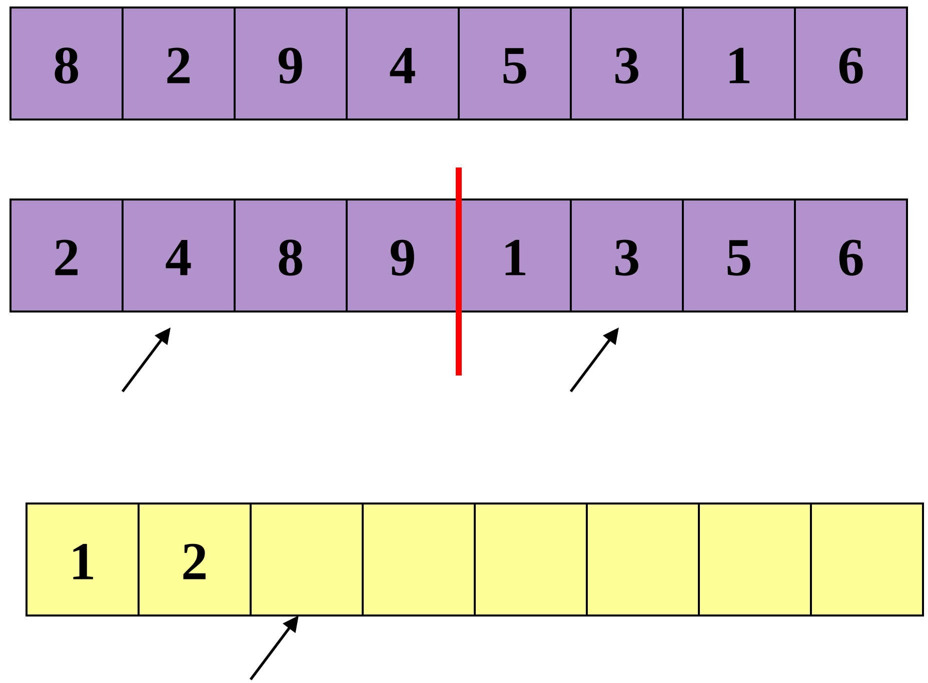
Start with:

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

Merge:

Use 3 “fingers”

and 1 more array



The Merging part

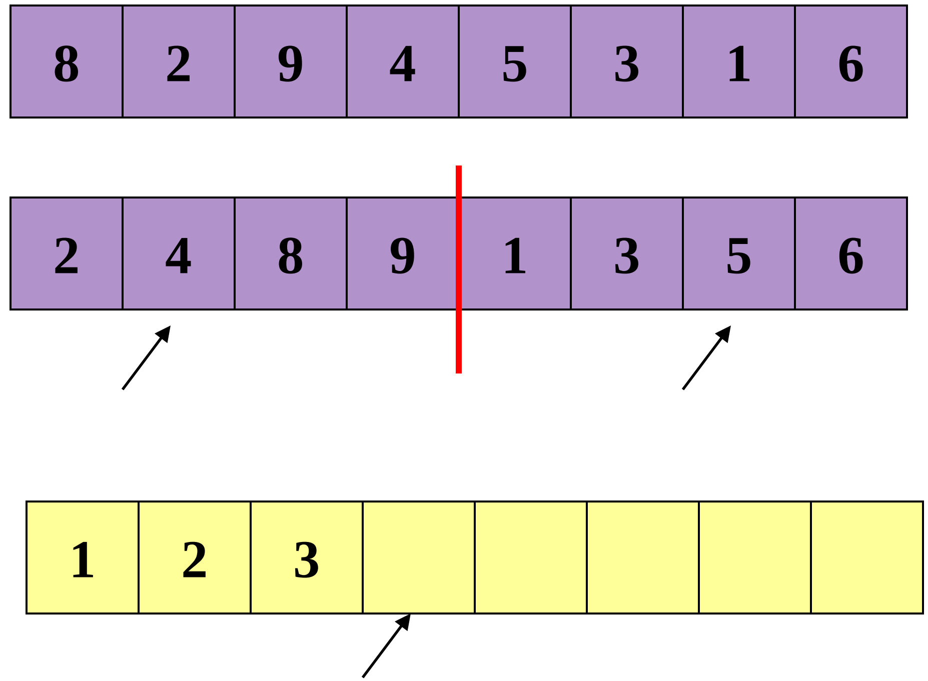
Start with:

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

Merge:

Use 3 “fingers”

and 1 more array



The Merging part

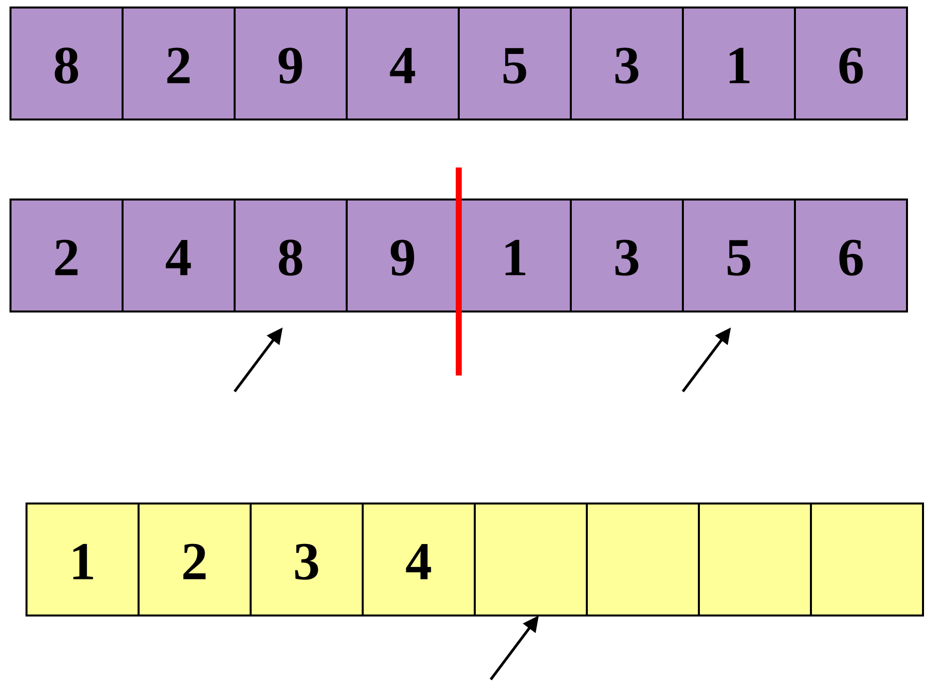
Start with:

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

Merge:

Use 3 “fingers”

and 1 more array



The Merging part

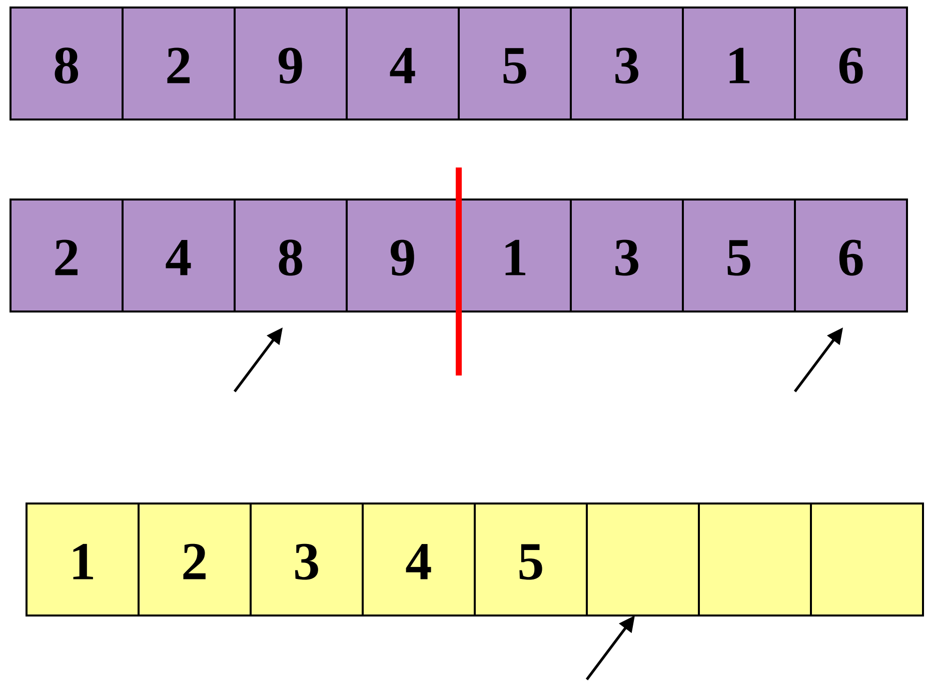
Start with:

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

Merge:

Use 3 “fingers”

and 1 more array



The Merging part

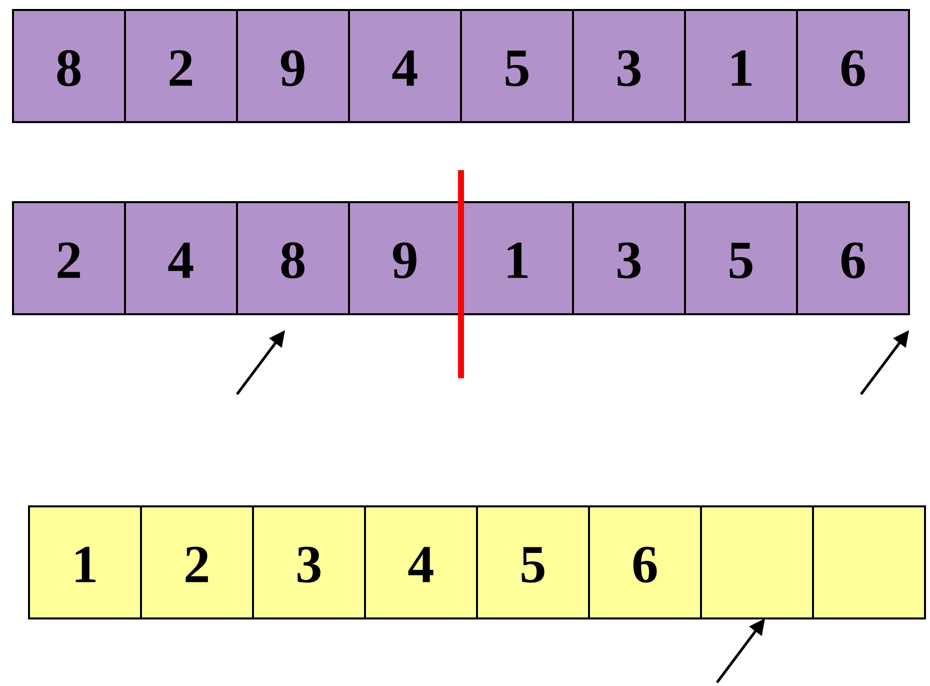
Start with:

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

Merge:

Use 3 “fingers”

and 1 more array



The Merging part

Start with:

8	2	9	4	5	3	1	6

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

2	4	8	9	1	3	5	6

Merge:

Use 3 “fingers”

and 1 more array

1	2	3	4	5	6	8	

The Merging part

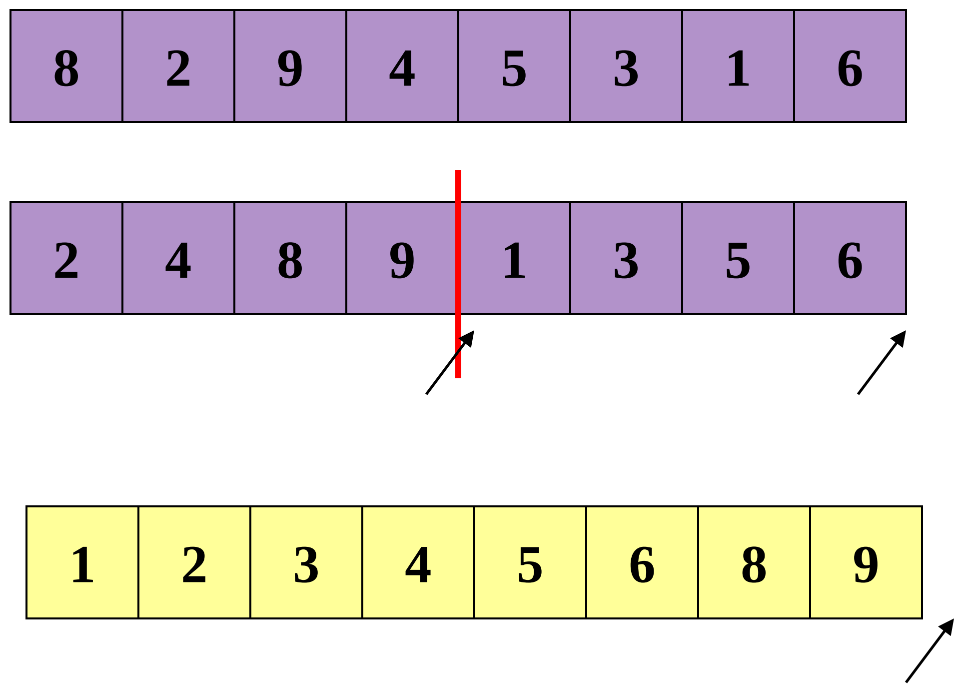
Start with:

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

Merge:

Use 3 “fingers”

and 1 more array



The Merging part

Start with:

8	2	9	4	5	3	1	6

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

2	4	8	9	1	3	5	6

Merge:

Use 3 “fingers”

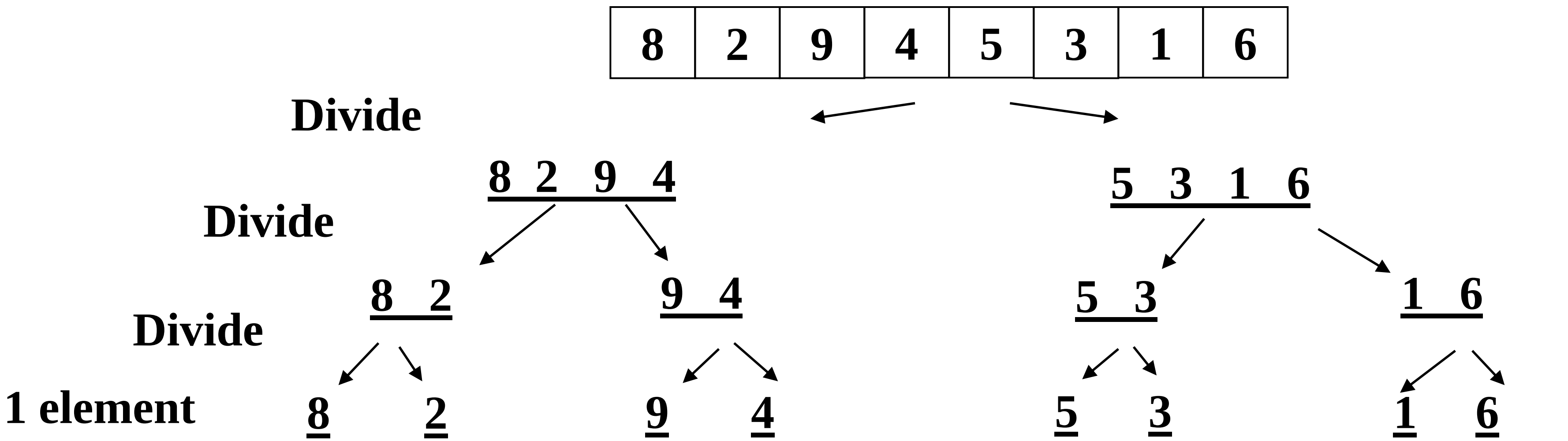
and 1 more array

1	2	3	4	5	6	8	9

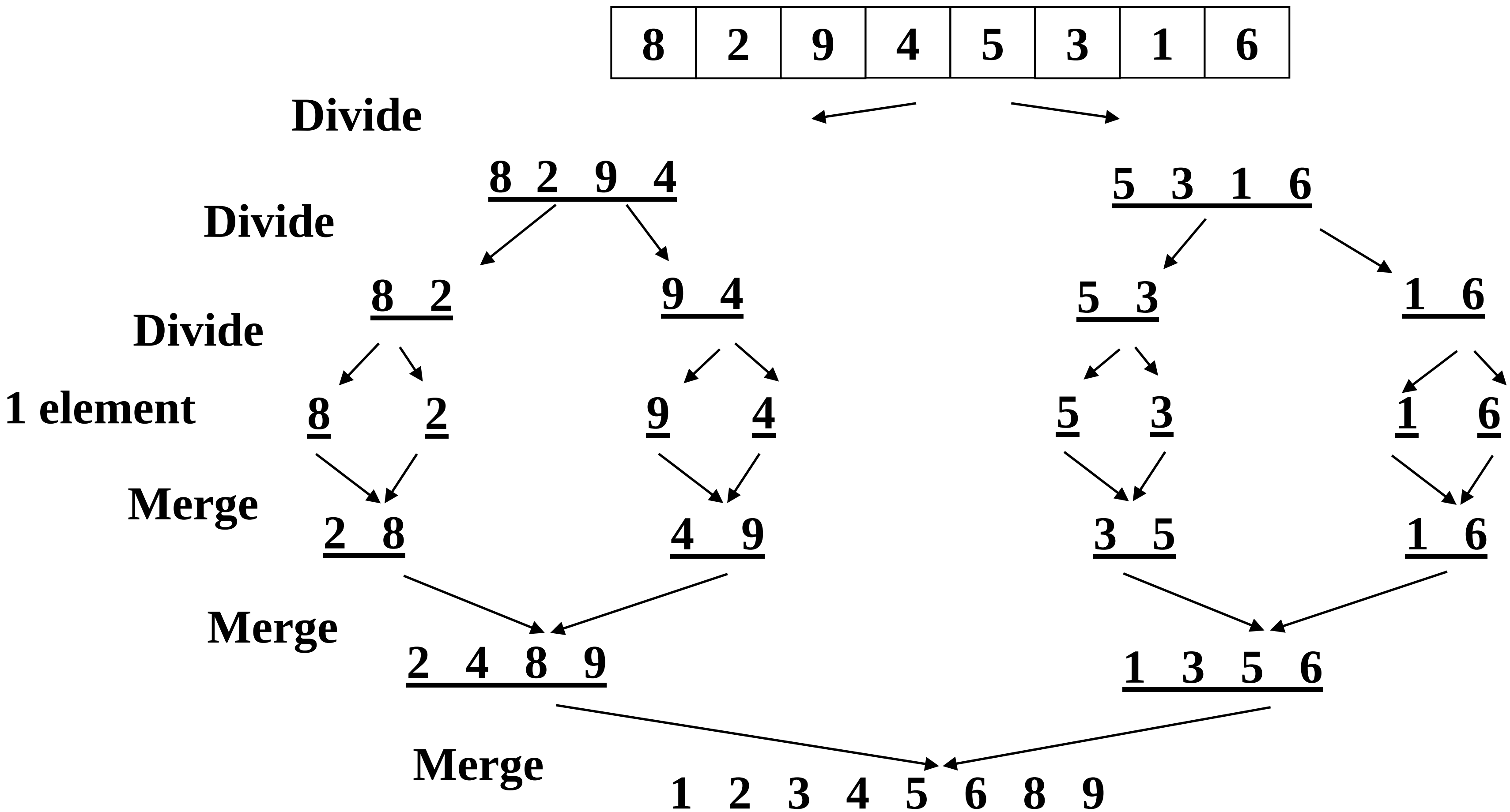
After merge, copy back to original array

1	2	3	4	5	6	8	9

Mergesort example: Recursively splitting list in half

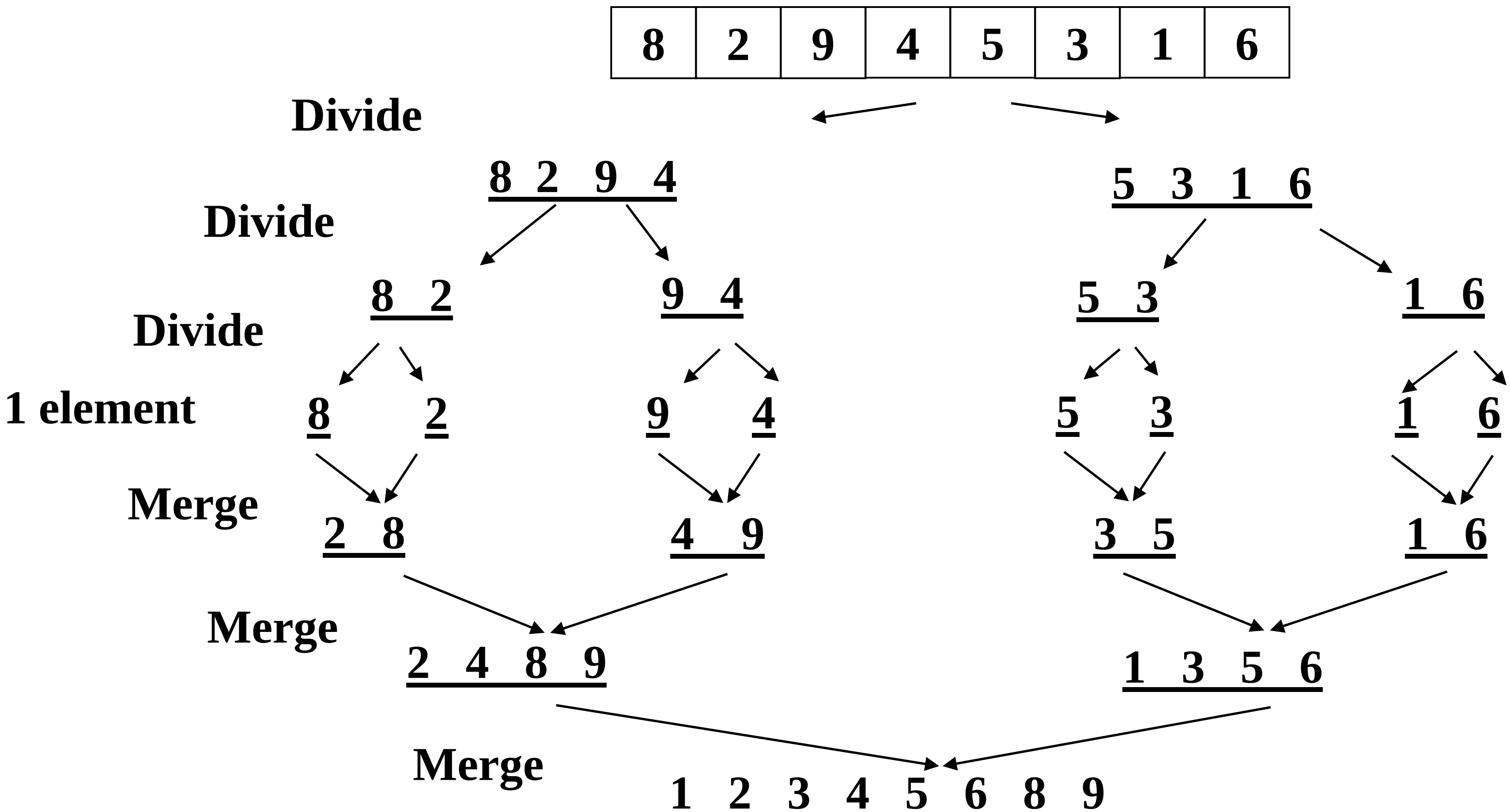


Mergesort example: Merge as we return from recursive calls



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

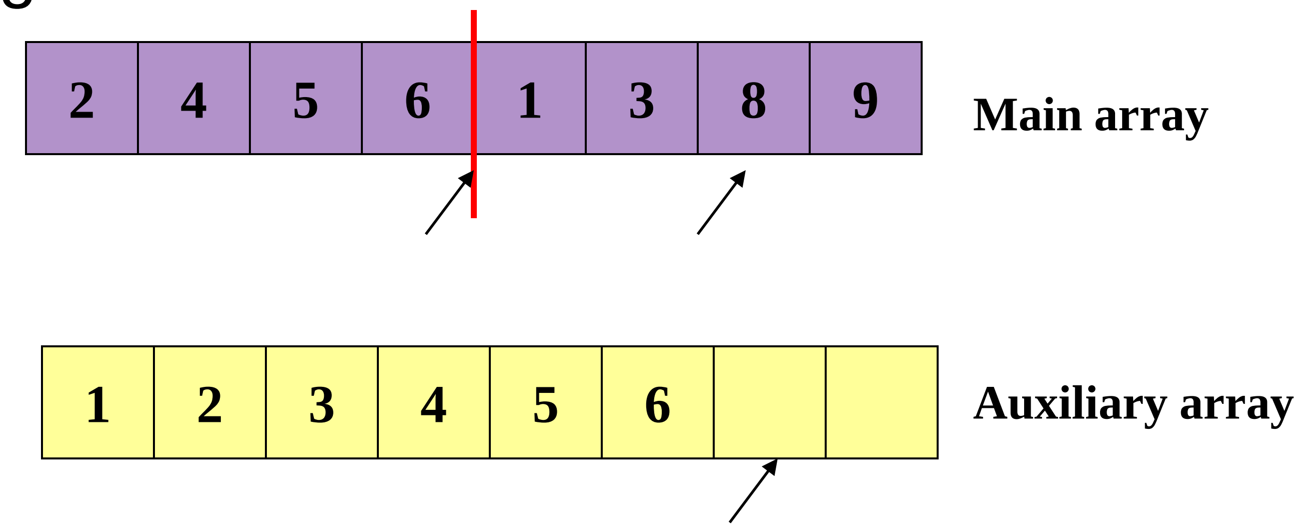
Mergesort example: Merge as we return from recursive calls



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

Some details: saving a little time

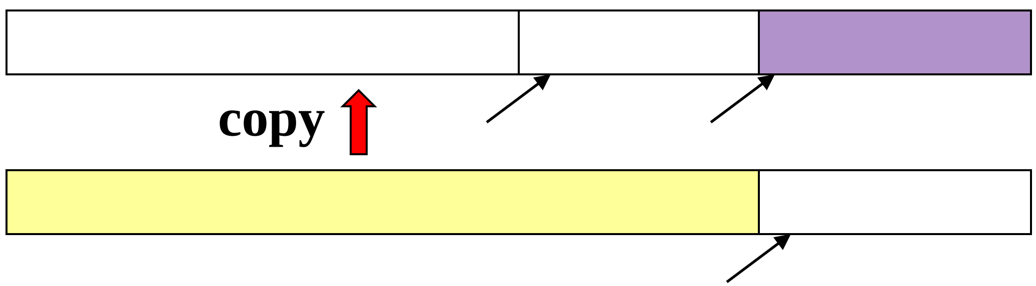
- What if the final steps of our merging looked like the following:



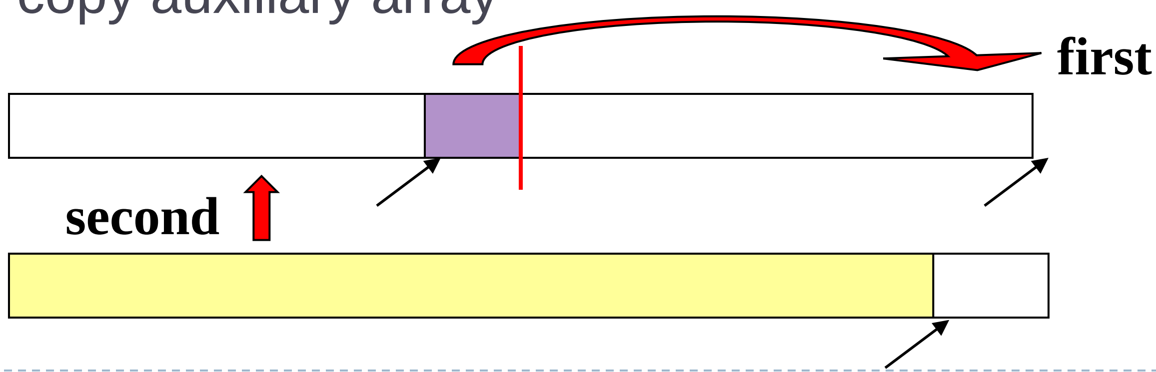
- Seems kind of wasteful to copy 8 & 9 to the auxiliary array just to copy them immediately back…

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):

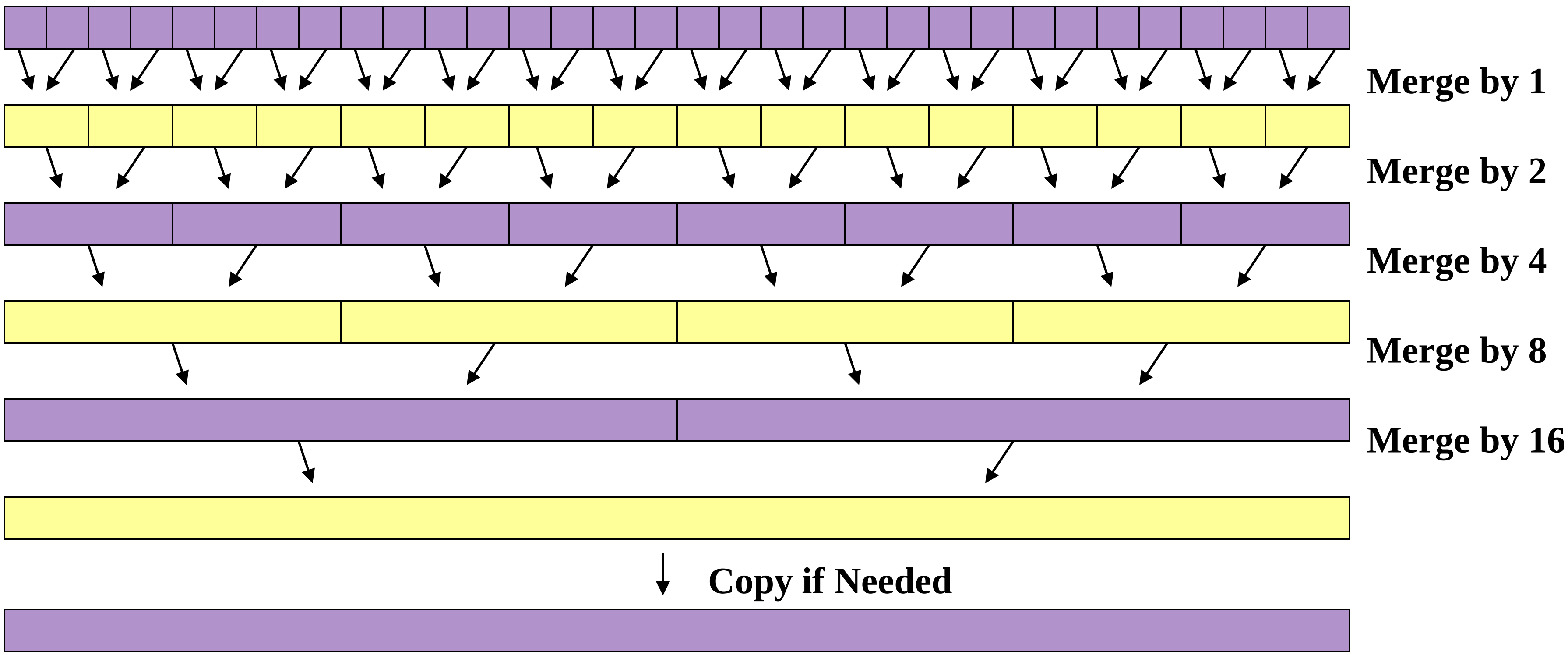
Don't copy back – at 2nd, 4th, 6th, … merging stages, use the original array as the auxiliary array and vice-versa

- 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

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

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 sub-problems of size $n/2$ and then an $O(n)$ merge

Recurrence relation:

$$T(1) = c_1$$

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

MergeSort Recurrence:
$T(1) = c_1$
$T(n) = 2T(n/2) + c_2 n$

MergeSort Recurrence

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

$T(1) = 1$, $T(n) = 2T(n/2) + n$; expand inner T()

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

$= 4(2T(n/8) + n/4) + 2n = 8T(n/8) + 3n$

….

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

How many expansions until we reach the base case?

$n/2^k=1$, so $n=2^k$, so $k=\log_2 n$

So $T(n)=2^{\log 2n}\ T(1)+n\log_2 n = nT(1)+n\log_2 n$

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

MergeSort Recurrence:

$T(1) = c_1$

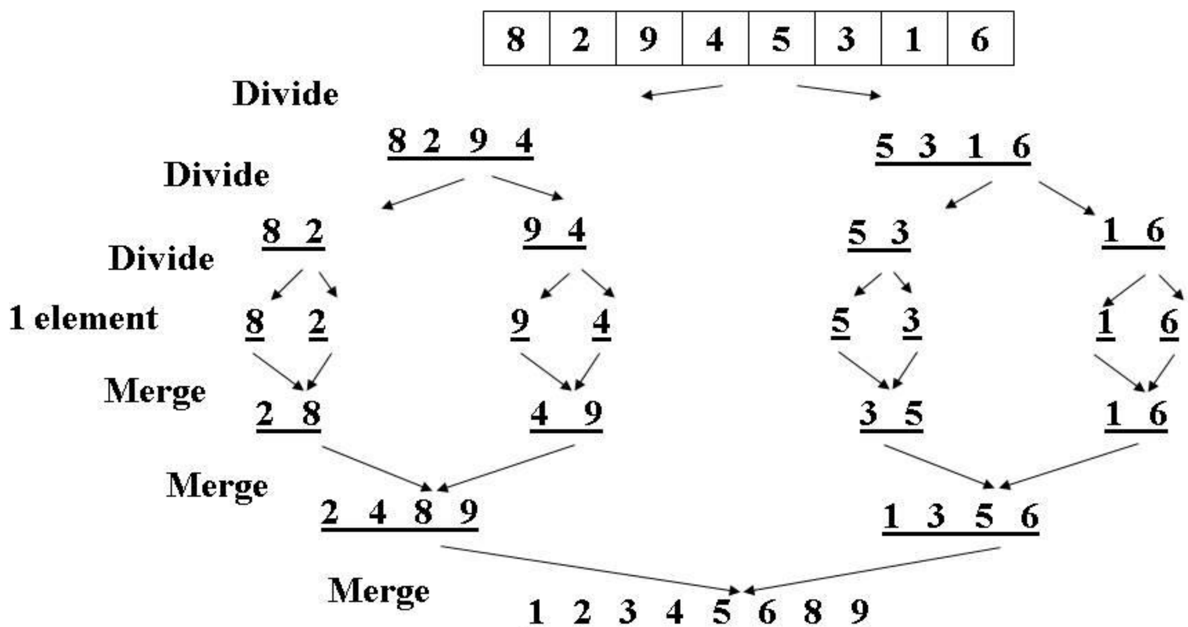
$T(n) = 2T(n/2) + c_2 n$

Or more intuitively…

This recurrence comes up frequently; good to memorize as $O(n \log n)$

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(nlogn)
  - So why use QuickSort?

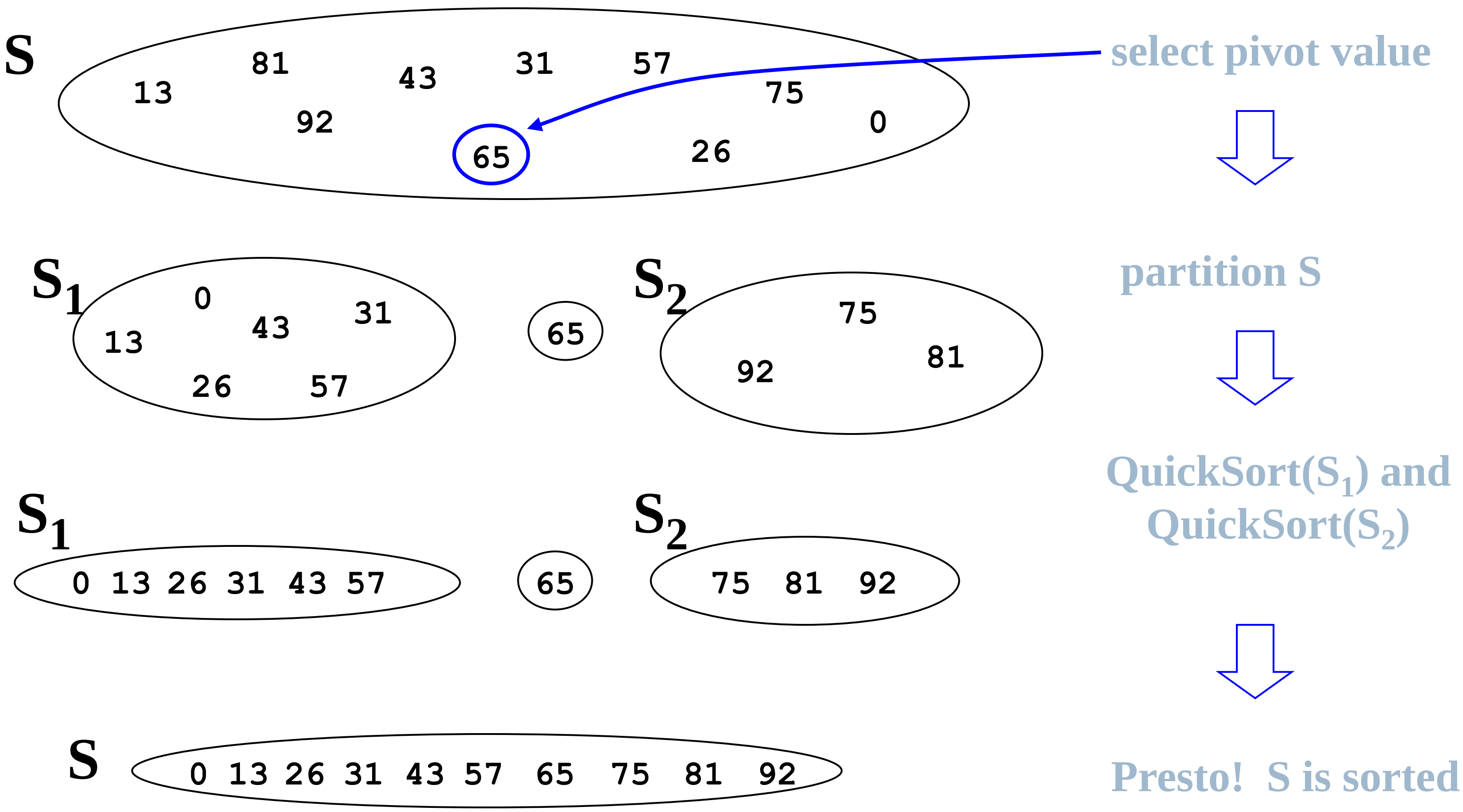
- Can be faster than mergesort
  - Often believed to be faster
  - Does fewer copies and more comparisons, so it depends on the relative cost of these two operations!

QuickSort overview

- 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
- Partition all the data into:
  - The elements less than the pivot
  - The pivot
  - The elements greater than the pivot
- Ex: Say we have 8, 4, 2, 9, 3, 5, 7
  - Say we pick '5' as the pivot
  - Left half (in no particular order): 4, 2, 3
  - Right half (in no particular order): 8, 9, 7
  - Result of partitioning: 4, 2, 3, 5, 8, 9, 7

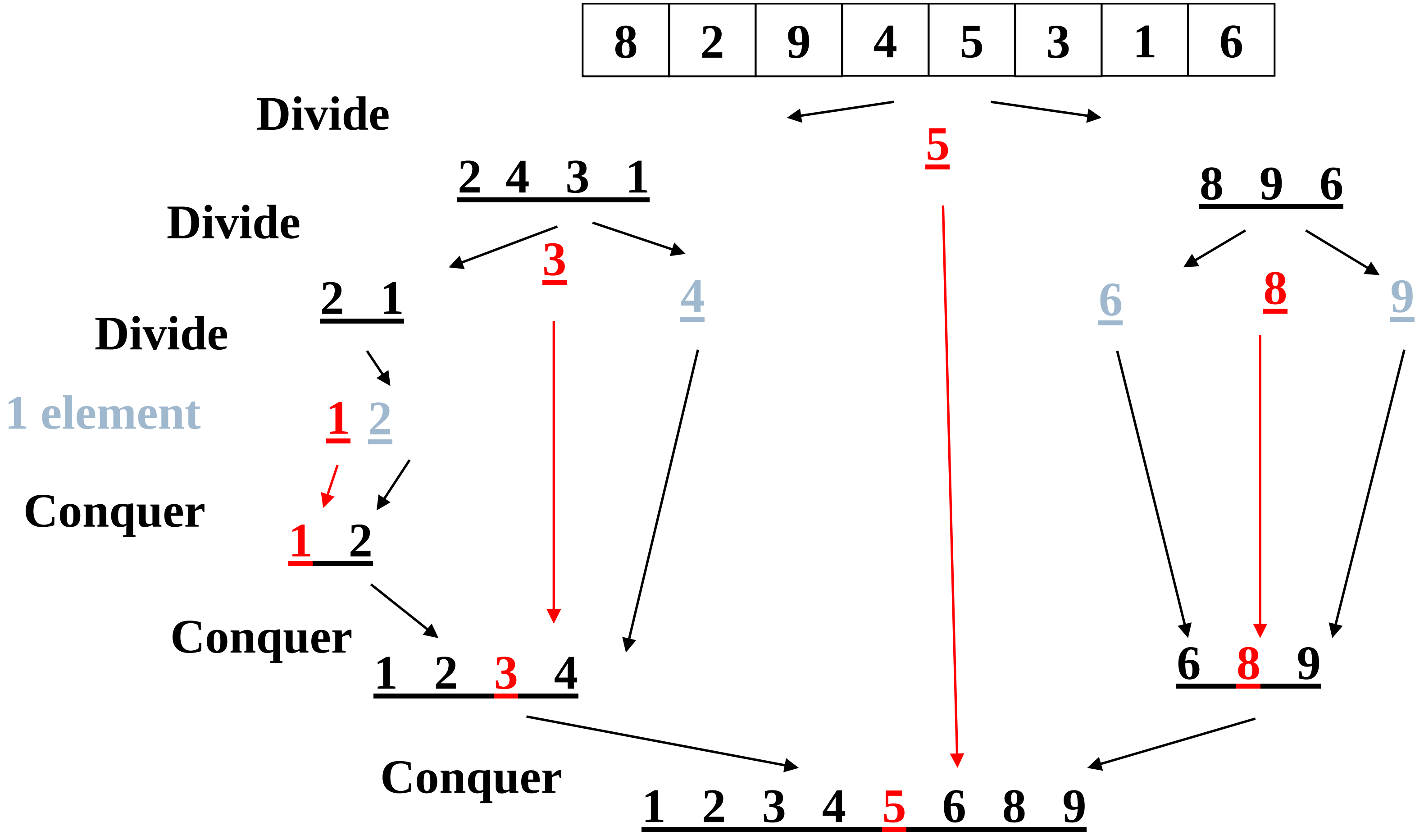
- That's great and all… but not really in order…

Think in terms of sets

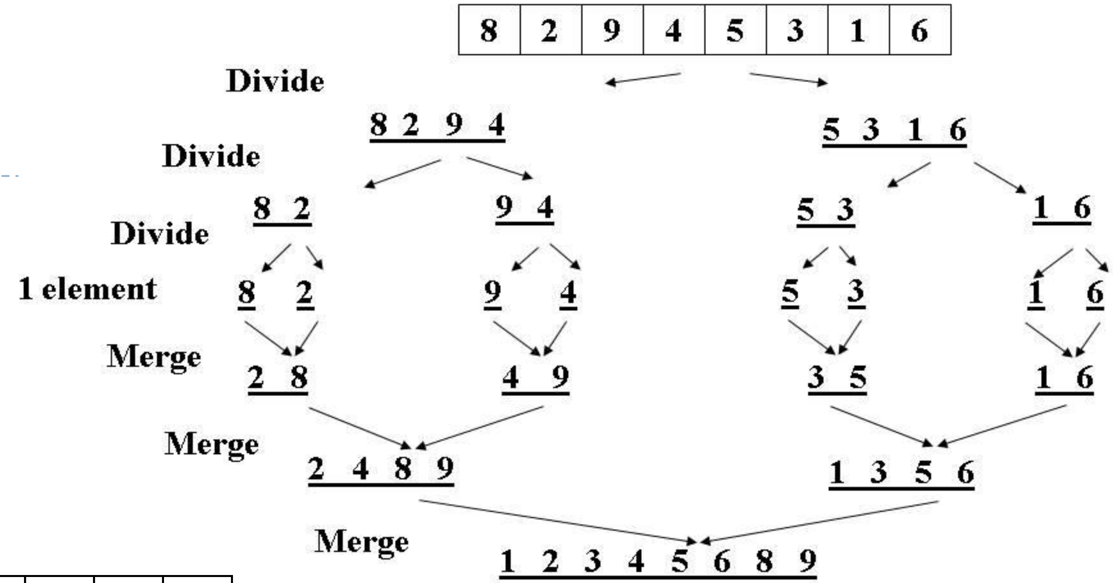


[Weiss]

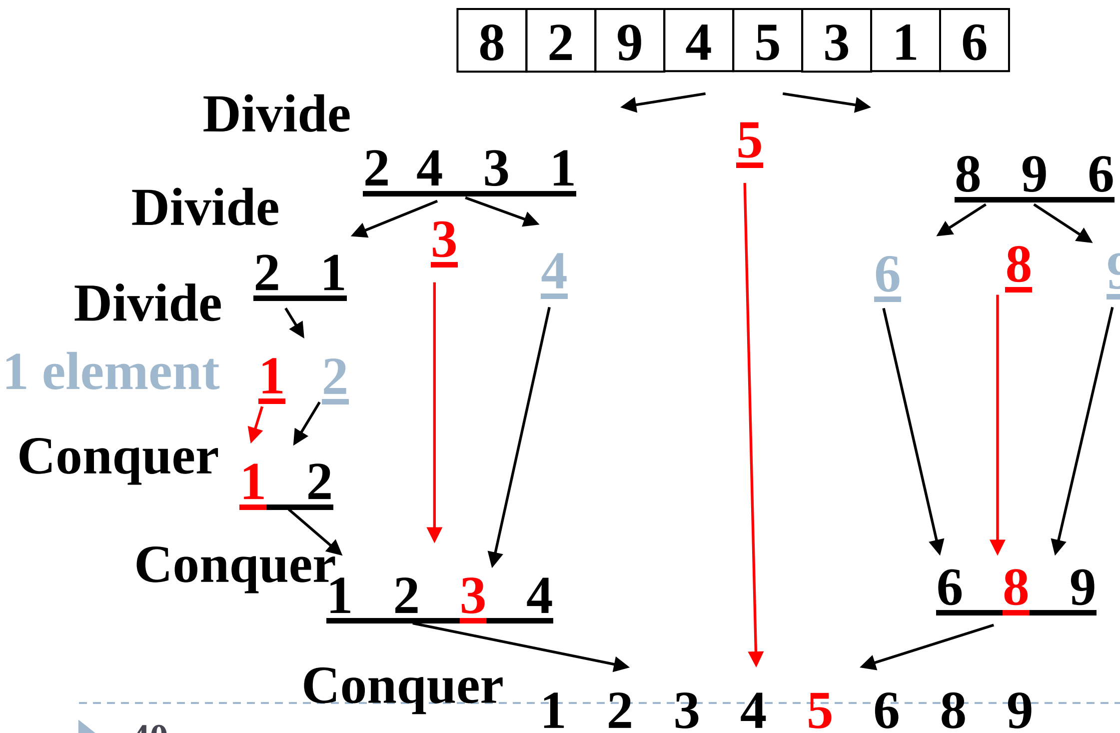
QuickSort Recursion Tree



MergeSort Recursion Tree



QuickSort Recursion Tree

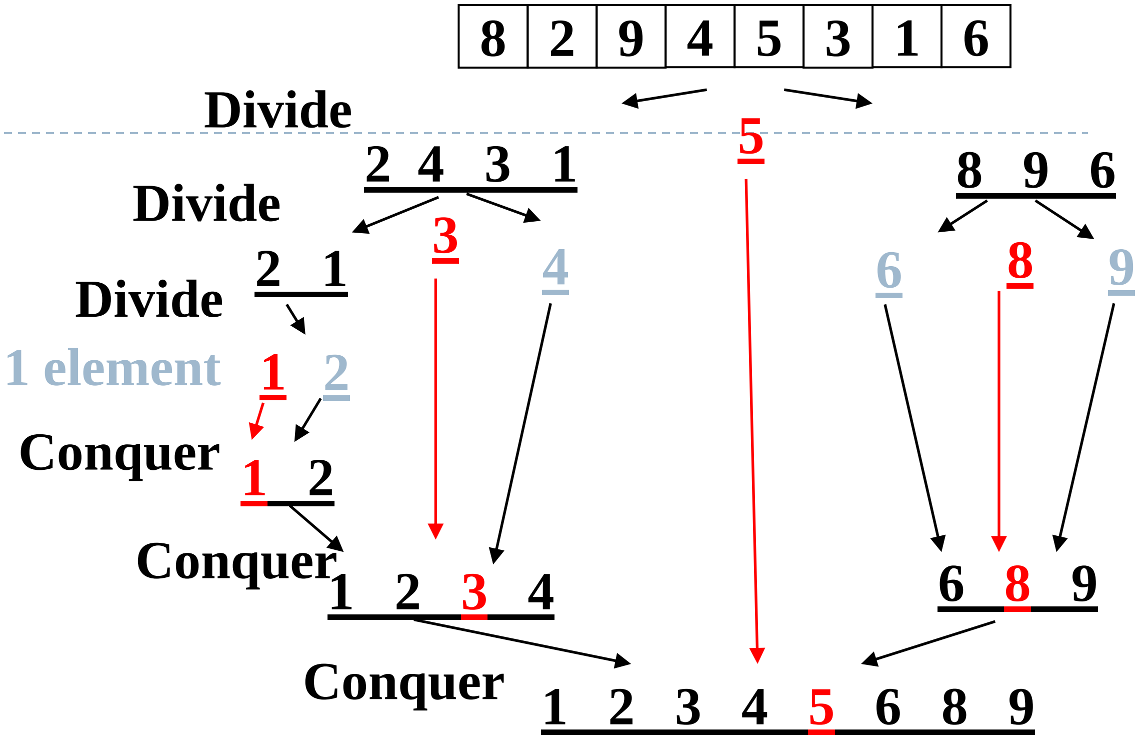


Details

We haven't 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
  - $O(n^2)$

Potential pivot rules

Say we call

Quicksort(int[] arr,int lo,int hi)

To sort arr from [lo,hi) (including lo, excluding hi)

- How about picking **`arr[lo]?`**
  - Quick to pick pivot, but worst-case is (mostly) sorted input
  - Same for picking arr[hi-1]
- How about picking random element in the range?
  - Does as well as any technique, but (pseudo)random number generation can be slow
  - Still probably not a bad approach
- Median of 3
  - Pick median of **`arr[lo], arr[hi-1], arr[(hi+lo)/2]`**
  - Common heuristic that tends to work well
  - Can still give us worst case though

Partitioning

- That is, given 8, 4, 2, 9, 3, 5, 7 and pivot 5
  - Getting 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

Partitioning 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

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

Analysis

- Best-case: Pivot is always the median: Halve each time

  T(0)=T(1)=1

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

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

- Worst-case: Pivot is always smallest or largest element: Reduce size by 1 each time

  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*
  - 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
  - None of this affects asymptotic complexity

Cutoff skeleton

Here the range is [lo,hi)

```
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; most nodes will be at those bottom layers