

Sorting



A Useful Invariant

- Binary Search only works **if the array is sorted**
- BSTs are **based around the idea of sorting the input**

“Local” vs. “Global” Views of Data

- All of our data structure so far only gave us a local view:
 - Heaps gave us a view of the max or min
 - Stacks and Queues gave us a view of most/least recent
 - Dictionaries give us a view of “associated data”
- A “global” view tells us how the elements all interact with each other
- There is no “best” sorting algorithm: most sorts have a purpose

SORT is the computational problem with the following requirements:

Inputs

- An array A of E data of length L .
- A consistent, total ordering on all elements of type E :
 $\text{compare}(a, b)$

Post-Conditions

- For all $0 \leq i < j < L$, $A[i] \leq A[j]$
- Every element originally in the array must be somewhere in the resulting array.

An algorithm that solves this computational problem is called a **Comparison Sort**.

There are several important properties sorting algorithms

Definition (In-Place Sorting)

A sorting algorithm is **in-place** if we don't require (more than $\mathcal{O}(1)$) extra space to do the sort.

It's a useful property, because:

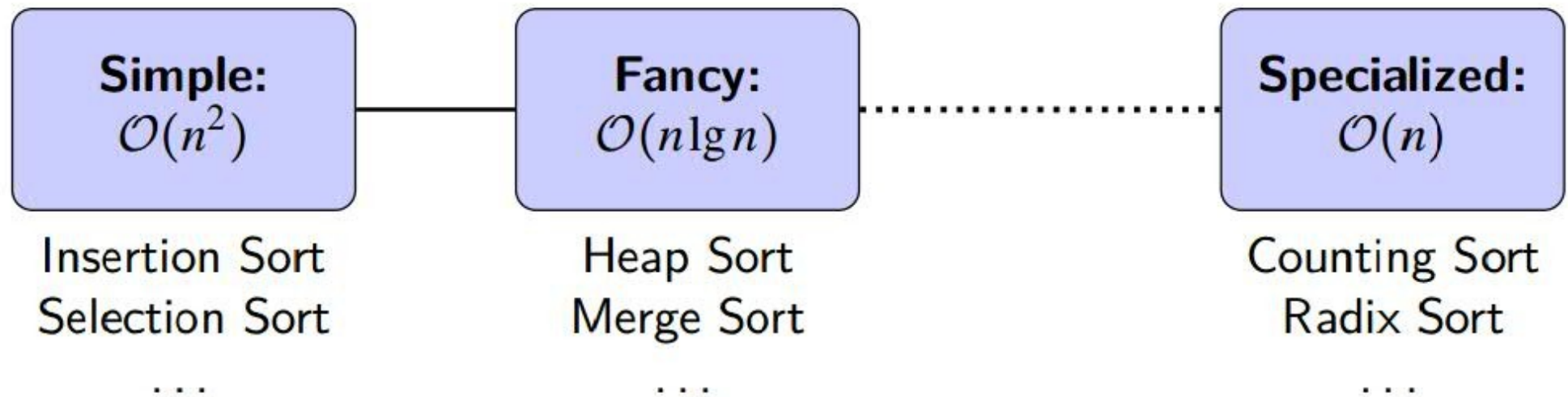
- The less memory we use the better...

Definition (Stable Sorting)

A sorting algorithm is **stable** if the order of any **equal** elements remains the same.

It's a useful property, because:

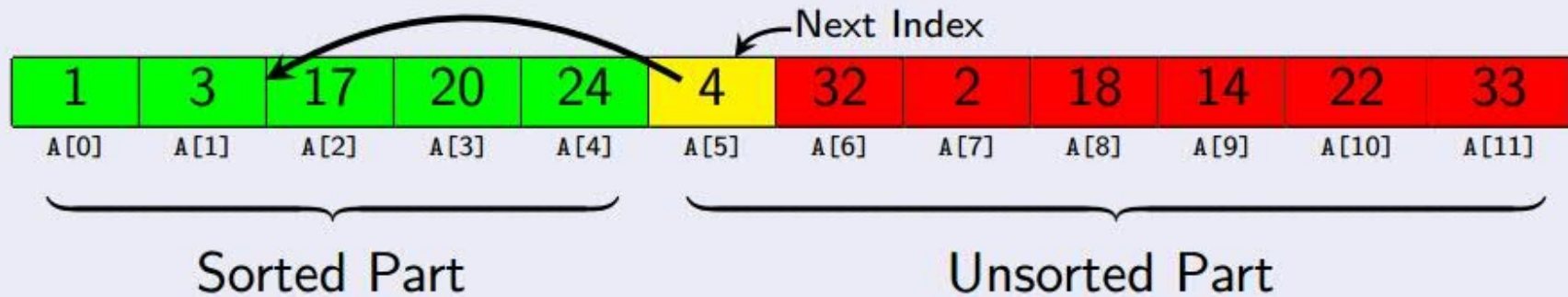
- We often want to first sort by one index and then another.
- Two objects might be equal but not completely duplicates.



There are a lot of different sorting algorithms out there!

We're not going to cover **all** of them, but we will cover the ones that demonstrate clear advantages in one way or another.

Insertion Sort



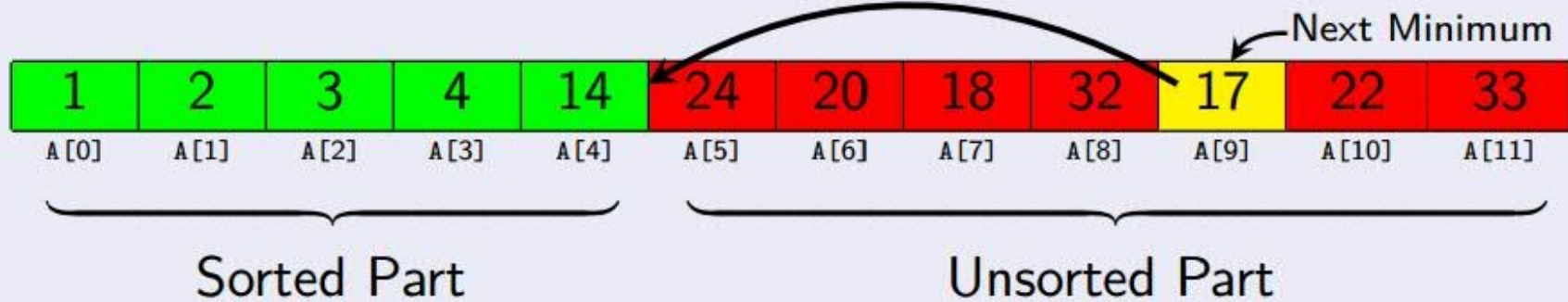
Algorithm

```
1 // i is "# of elements sorted"
2 for (i = 0; i < n; i++) {
3     // Scan from i - 1 to 0 to
4     // find the correct location
5 }
```

Runtime and Analysis

- Best Case?
- Average Case?
- Worst Case?
- In-Place?
- Stable?

Selection Sort



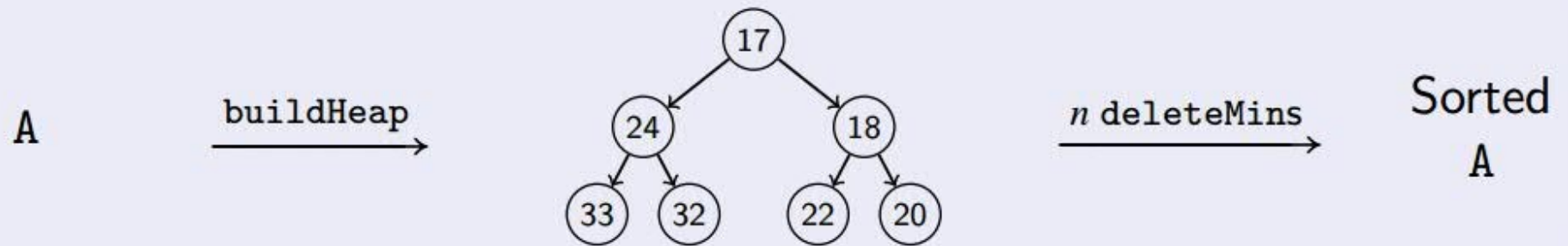
Algorithm

```
1 // i is "# of elements sorted"
2 for (i = 0; i < n; i++) {
3     findMin(i, n);
4     swap(i, j);
5 }
```

Runtime and Analysis

- Best Case?
- Average Case?
- Worst Case?
- In-Place?
- Stable?

Heap sort



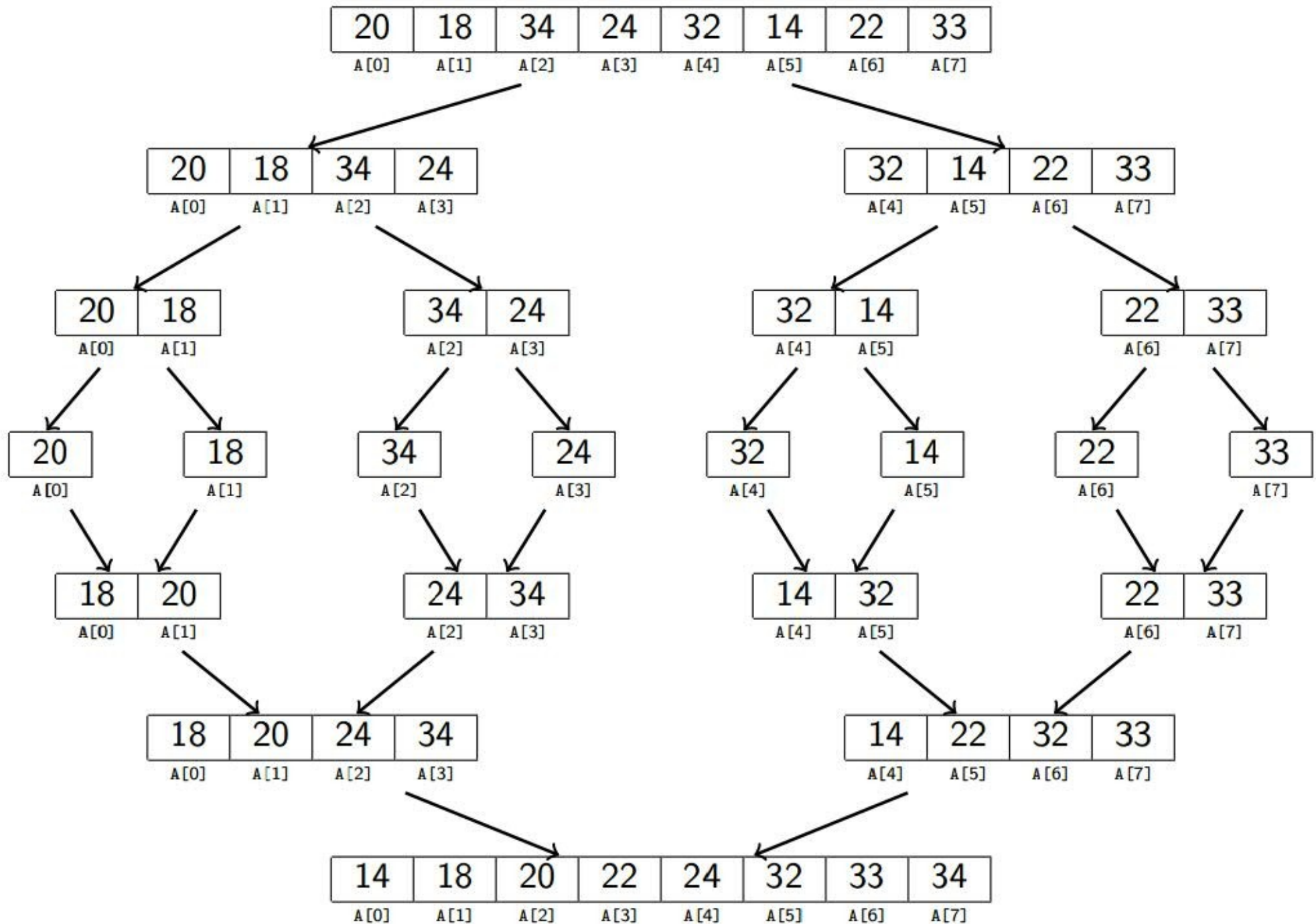
Algorithm

```
1 E[] A = buildHeap();
2 for (i = 0; i < n; i++) {
3     int min = A.deleteMin();
4     // Put min in array
5 }
```

Runtime and Analysis

- Best Case?
- Average Case?
- Worst Case?
- In-Place?
- Stable?

Merge Sort



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 example: Recursively splitting list in half

8	2	9	4	5	3	1	6
---	---	---	---	---	---	---	---

Divide



Divide

8 2 9 4

5 3 1 6

Divide

8 2

9 4

5 3

1 6

1 element

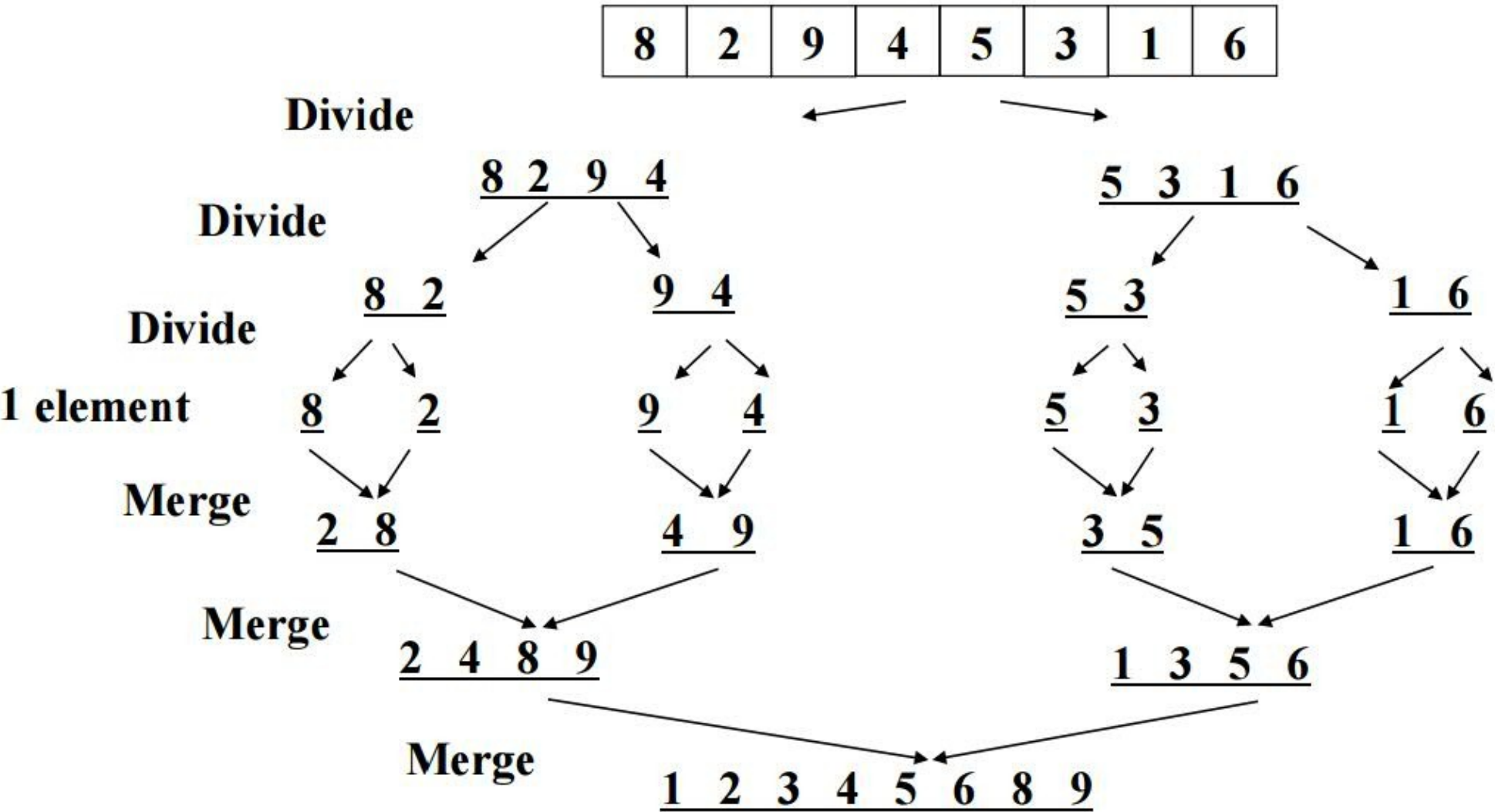
8 2

9 4

5 3

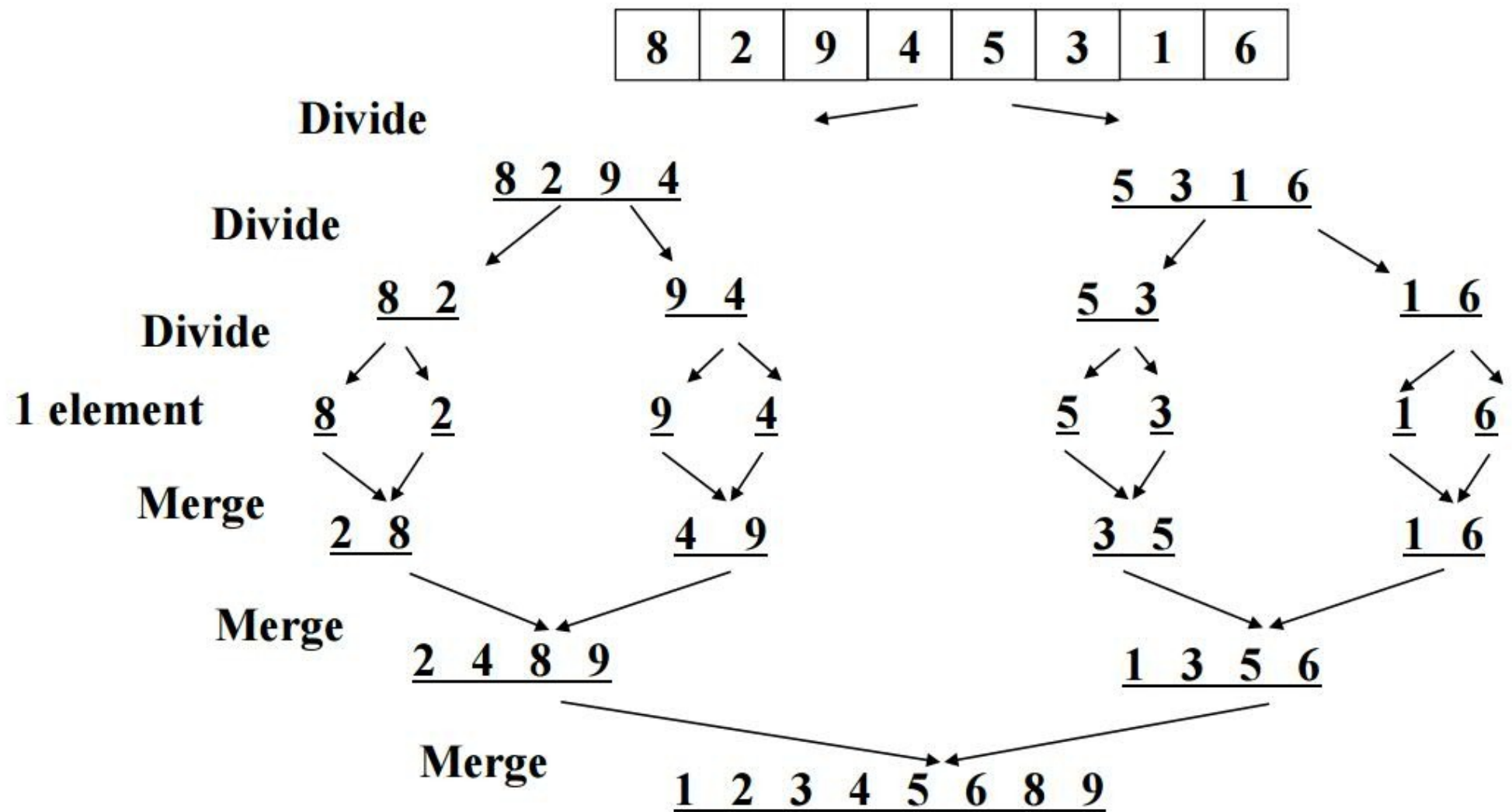
1 6

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

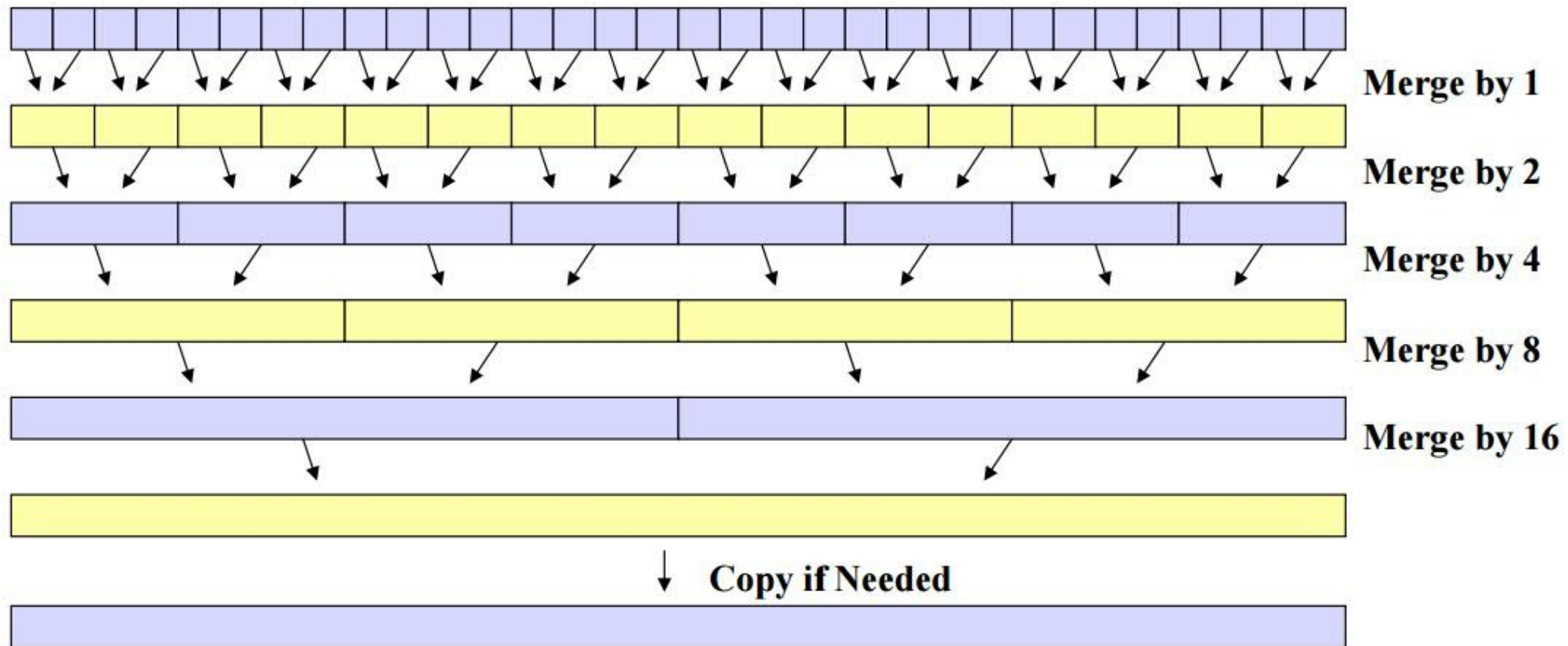


We need another array in which to do each merging step; merge results into there, then copy back to original array

*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

Quicksort

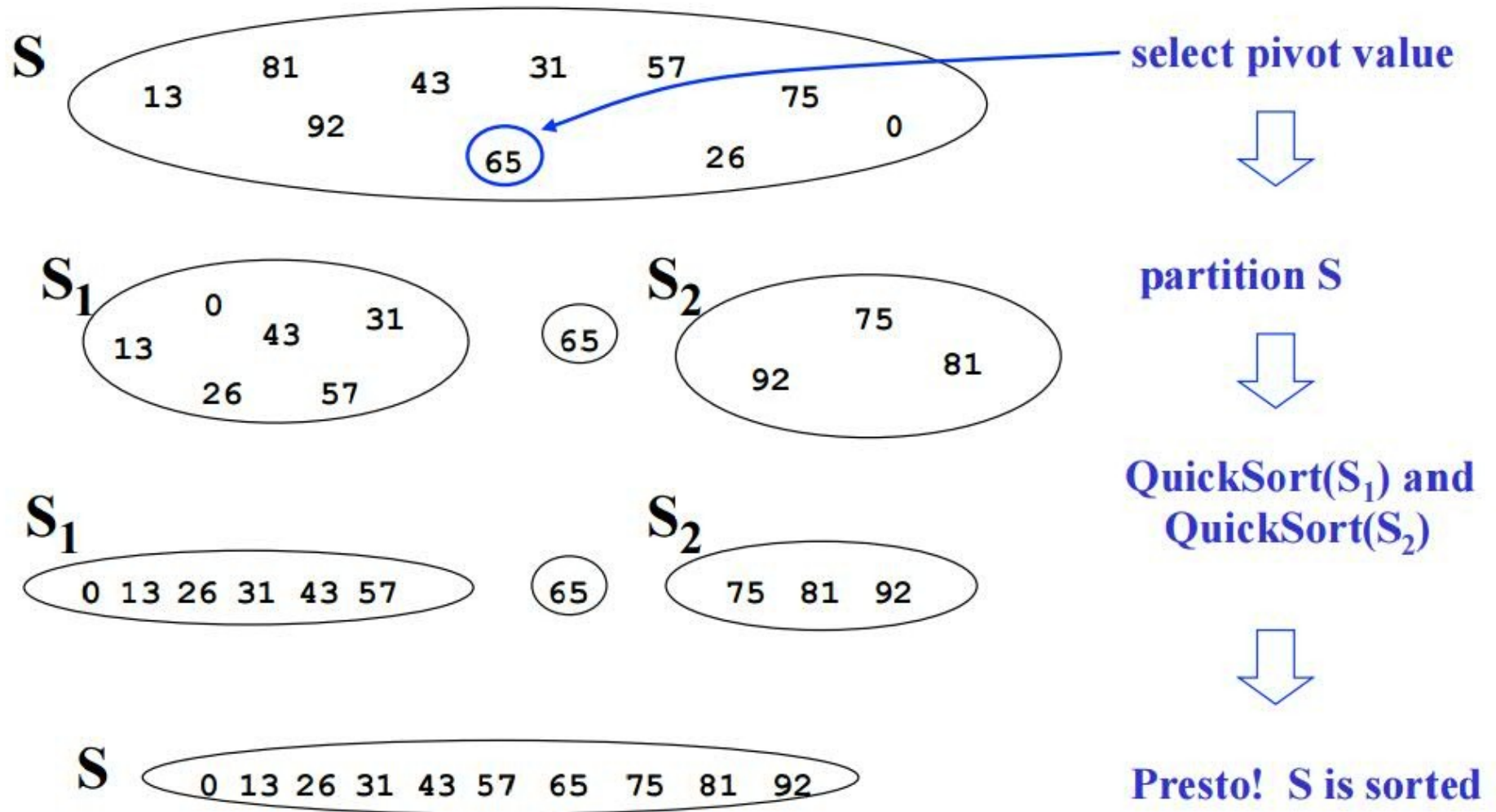
- Also uses divide-and-conquer
 - Recursively chop into halves
 - But, instead of doing all the work as we merge together, we'll do all the work as we recursively split into halves
 - Also unlike MergeSort, does not need auxiliary space
- $O(n \log n)$ on average 😊, but $O(n^2)$ worst-case 😞
 - MergeSort 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

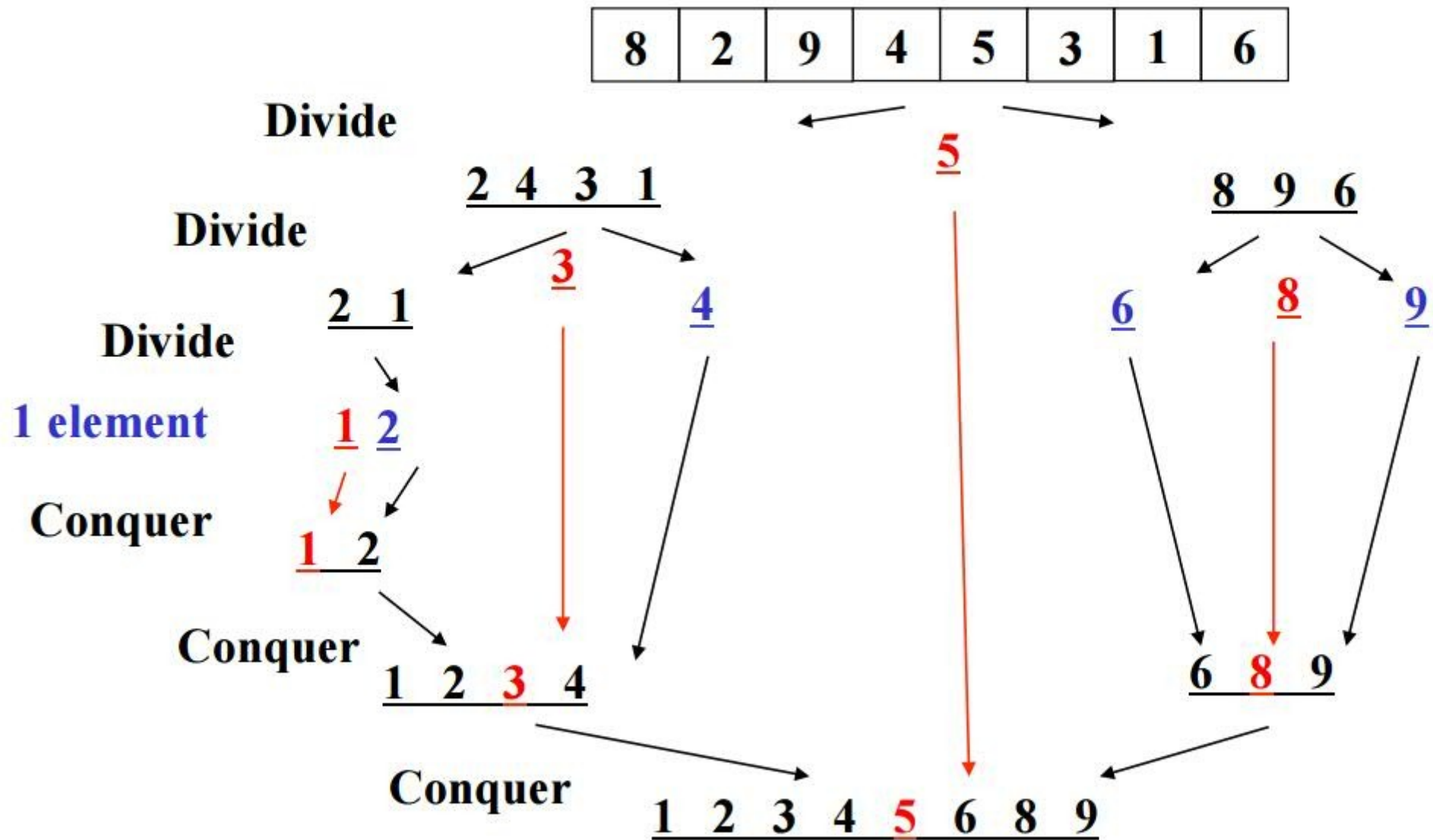
1. Pick a pivot element
 - Hopefully an element \sim median
 - Good QuickSort performance depends on good choice of pivot; we'll see why later, and talk about good pivot selection later
2. Partition all the data into:
 - A. The elements less than the pivot
 - B. The pivot
 - C. The elements greater than 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)

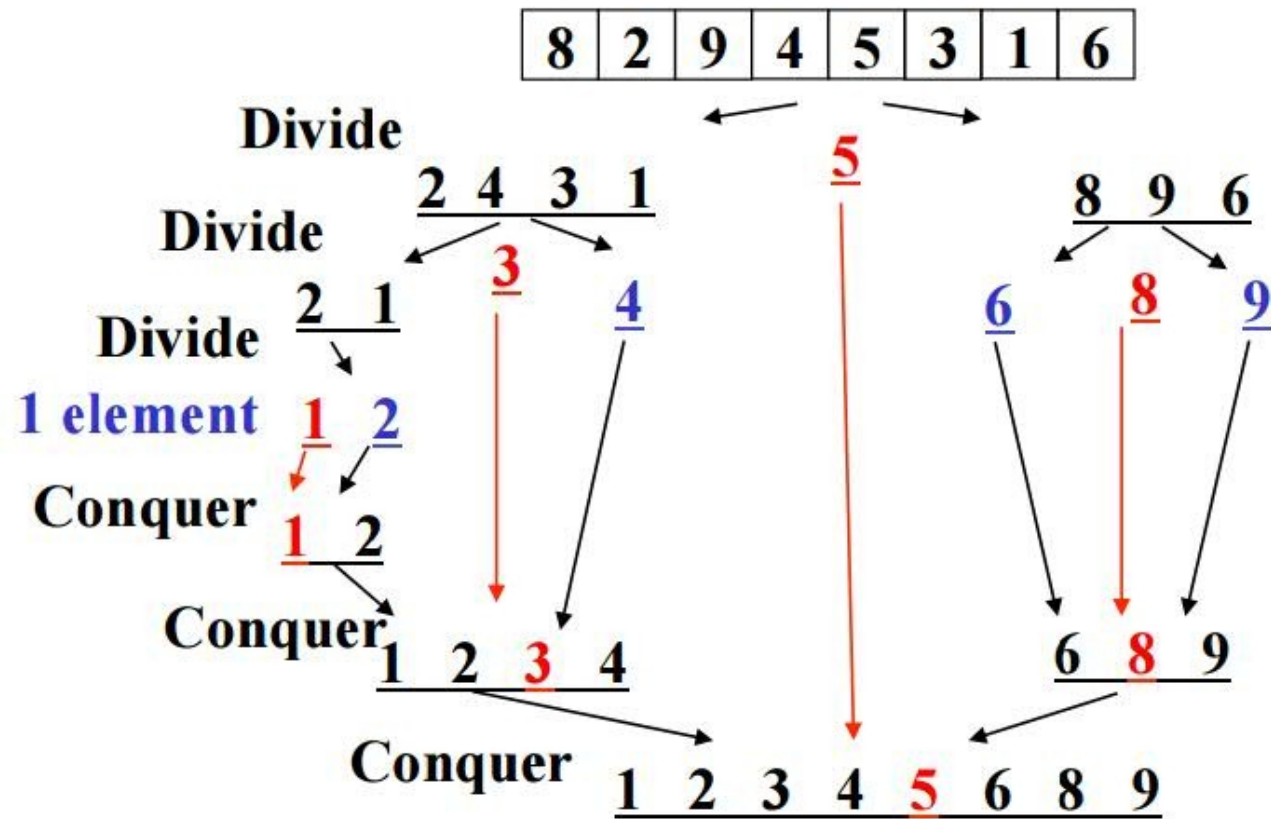
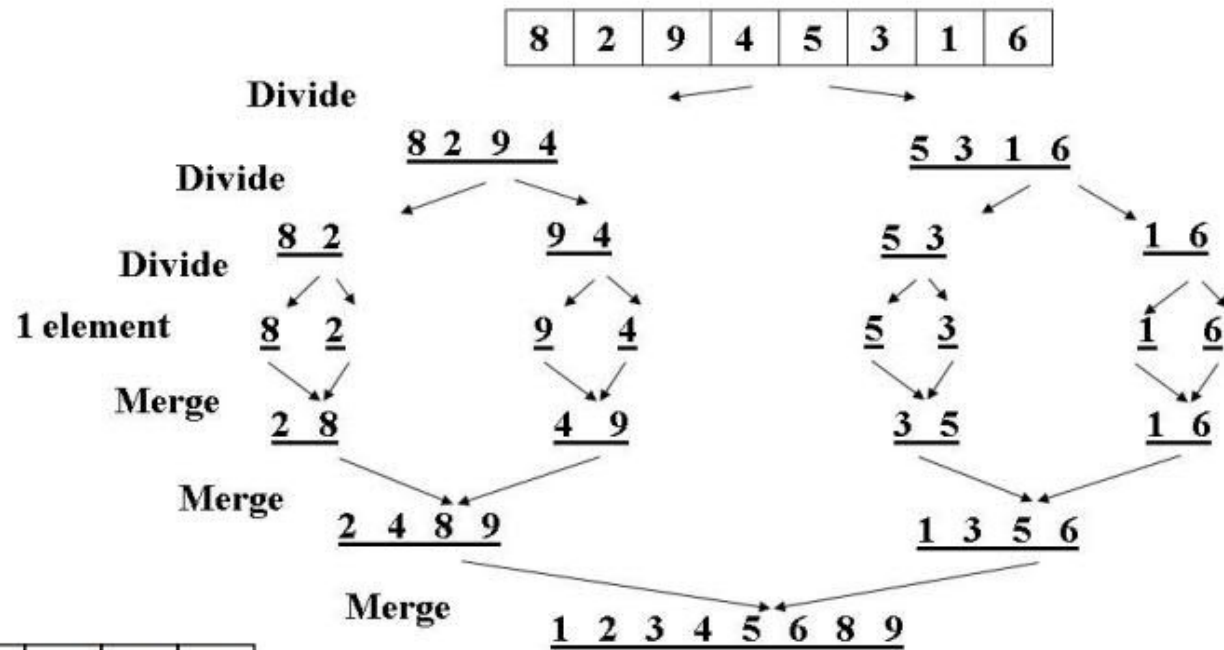
Quicksort: Think in terms of sets



Quicksort Example, showing recursion



MergeSort Recursion Tree



QuickSort Recursion Tree

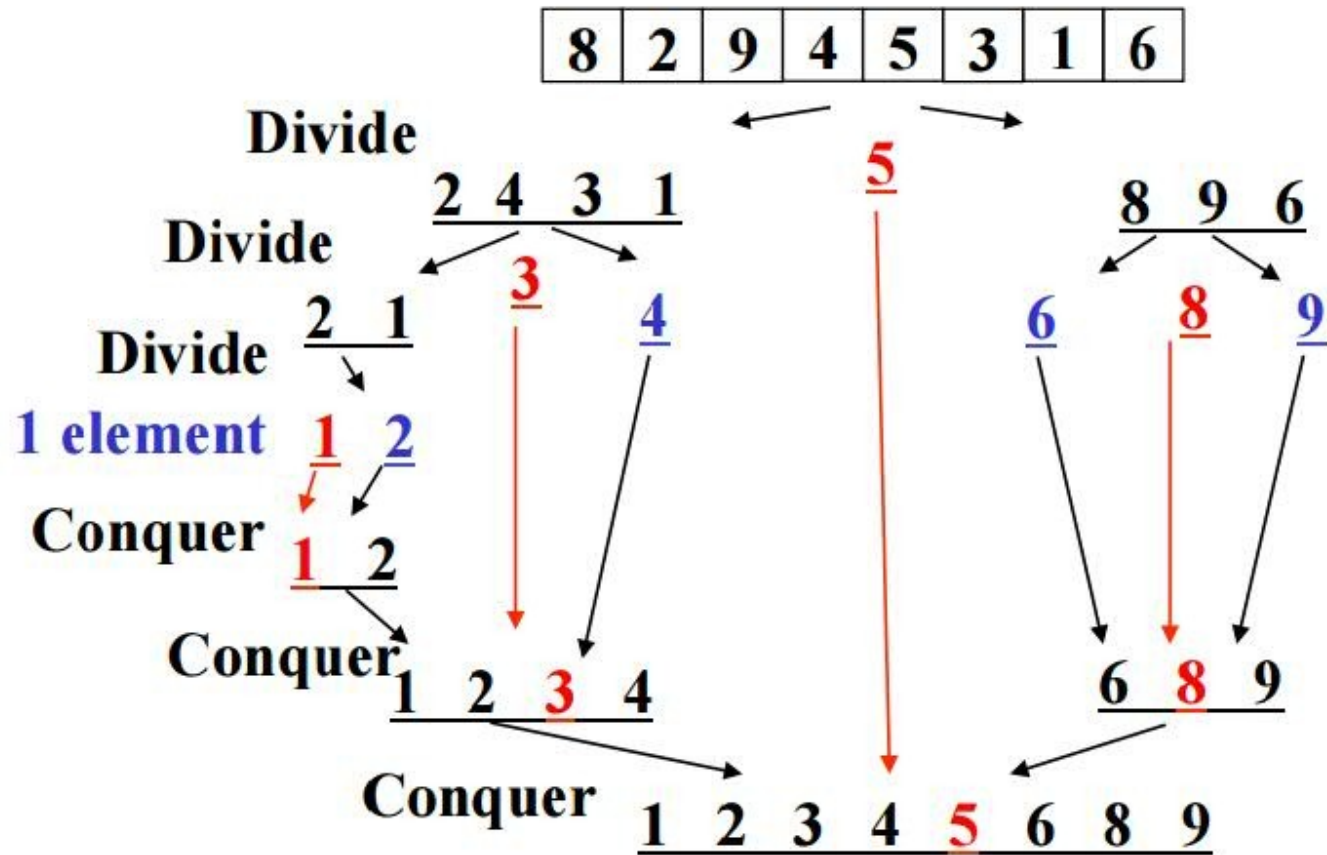
Quicksort Details

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

- Best pivot?
 - Median
 - Halve each time

- 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)...

- 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

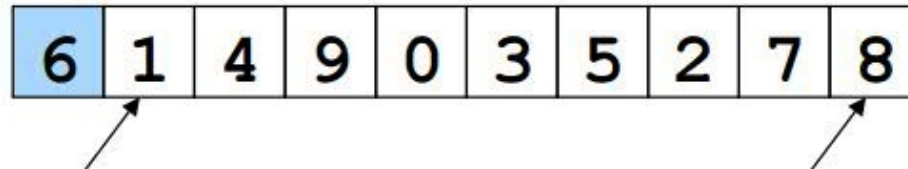
0	1	2	3	4	5	6	7	8	9
6	1	4	9	0	3	5	2	7	8



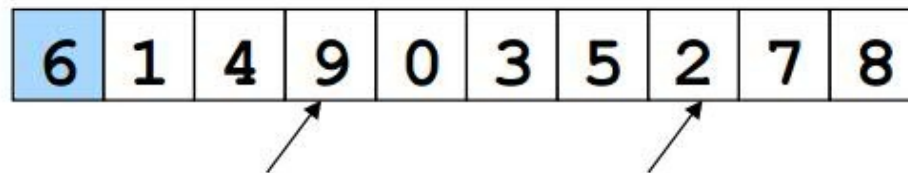
Quicksort Example

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

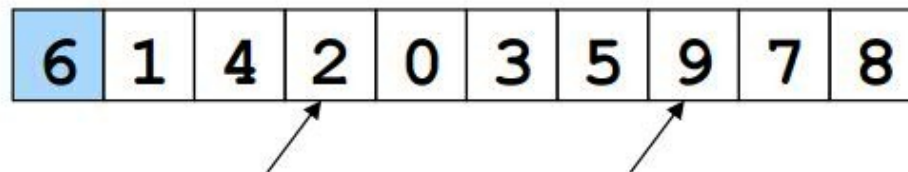
Now partition in place



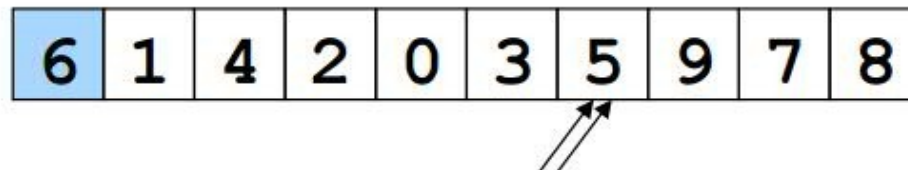
Move fingers



Swap



Move fingers



Move pivot



Quicksort Analysis

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

Quicksort 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 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)

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

