Sorting

Riley Porter

Introduction to Sorting

- Why study sorting?
  - Good algorithm practice!
- Different sorting algorithms have different trade-offs
  - No single "best" sort for all scenarios
  - Knowing one way to sort just isn't enough
- Not usually asked about on tech interviews...
  - but if it comes up, you look bad if you can't talk about it

More Reasons to Sort

General technique in computing:

Preprocess 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 elements in logarithmic time

Whether the performance of the preprocessing matters depends on

- How often the data will change (and how much it will change)
- How much data there is

Definition: Comparison Sort

A computational problem with the following input and output

Input:

An array **`A`** of length *n* comparable elements

Output:

The same array **`A`**, containing the same elements where:

for any **`i`** and **`j`** where $0 \leq$ **`i < j <`** ***n***

then **`A[i]`** $\leq$ **`A[j]`**

More Definitions

In-Place Sort:

A sorting algorithm is in-place if it requires only O(1) extra space to sort the array.

- Usually modifies input array
- Can be useful: lets us minimize memory

Stable Sort:

A sorting algorithm is stable if any equal items remain in the same relative order before and after the sort.

- Items that 'compare' the same might not be exact duplicates
- Might want to sort on some, but not all attributes of an item
- Can be useful to sort on one attribute first, then another one

Stable Sort Example

Input:

```
[(8, "fox"), (9, "dog"), (4, "wolf"), (8, "cow")]
```

Compare function: compare pairs by number only

Output (stable sort):

```
[(4, "wolf"), (8, "fox"), (8, "cow"), (9, "dog")]
```

Output (unstable sort):

```
[(4, "wolf"), (8, "cow"), (8, "fox"), (9, "dog")]
```

Lots of algorithms for sorting...

Quicksort, Merge sort, In-place merge sort, Heap sort, Insertion sort, Intro sort, Selection sort, Timsort, Cubesort, Shell sort, Bubble sort, Binary tree sort, Cycle sort, Library sort, Patience sorting, Smoothsort, Strand sort, Tournament sort, Cocktail sort, Comb sort, Gnome sort, Block sort, Stackoverflow sort, Odd-even sort, Pigeonhole sort, Bucket sort, Counting sort, Radix sort, Spreadsort, Burstsort, Flashsort, Postman sort, Bead sort, Simple pancake sort, Spaghetti sort, Sorting network, Bitonic sort, Bogosort, Stooge sort, Insertion sort, Slow sort, Rainbow sort...

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: $\Omega(n \log n)$

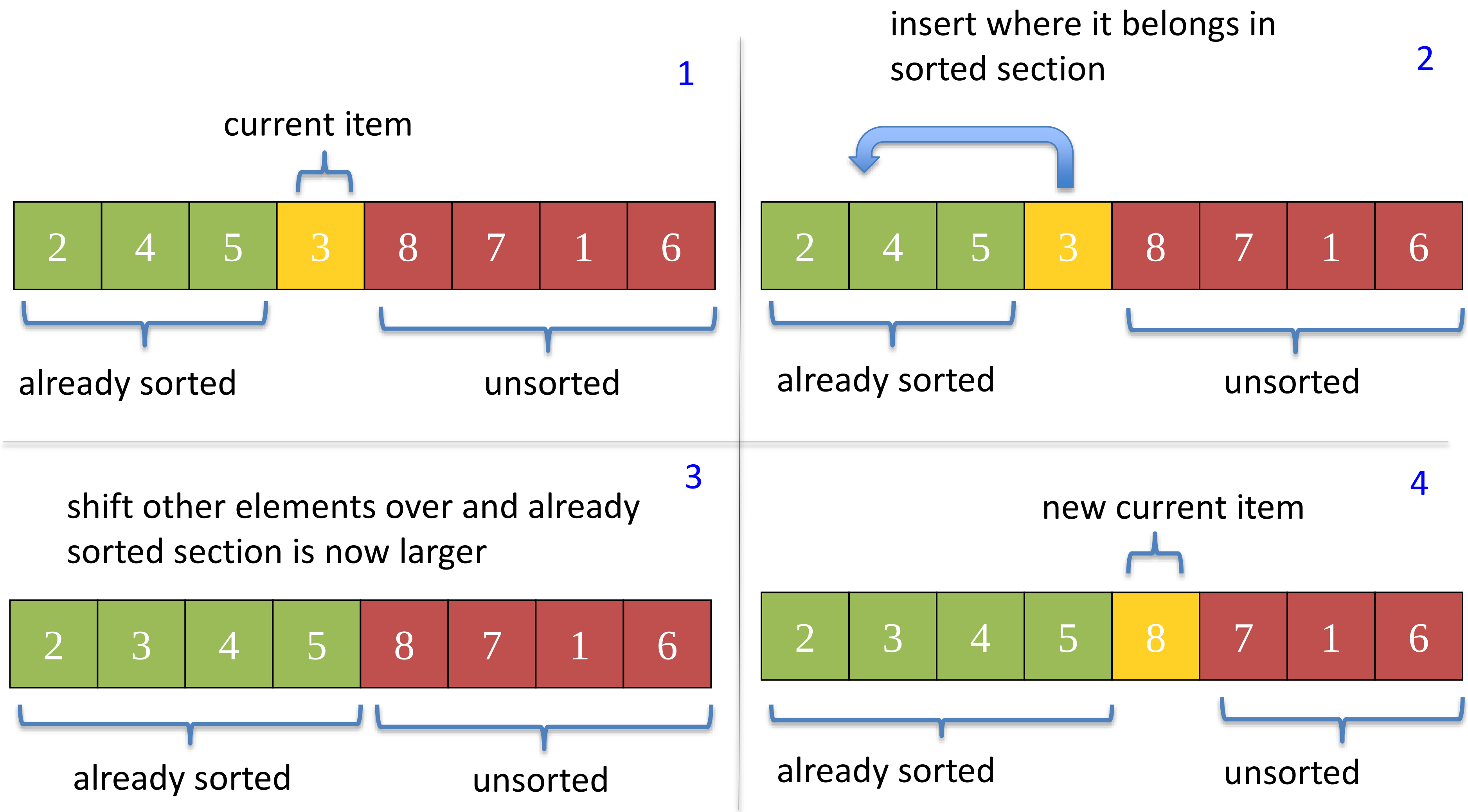
Specialized algorithms: $O(n)$

Bucket sort
Radix sort

Handling huge data sets

External sorting

Insertion Sort



Insertion Sort

- Idea: At step **k**, put the **k**th element in the correct position among the first **k** elements

```
for (int i = 0; i < n; i++) {
     // Find index to insert into
     int newIndex = findPlace(i);
     // Insert and shift nodes over
     shift(newIndex, i);
}
```

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

- Runtime?

  Best-case _____ Worst-case _____ Average-case ____

- Stable? _____ In-place? _____

Insertion Sort

- Idea: At step **`k`**, put the **`k`**th element in the correct position among the first **`k`** elements

```
for (int i = 0; i < n; i++) {
      // Find index to insert into
      int newIndex = findPlace(i);
      // Insert and shift nodes over
      shift(newIndex, i);
}
```

- **Loop invariant**: when loop index is **`i`**, first **`i`** elements are sorted from the first **`i`** elements in the array

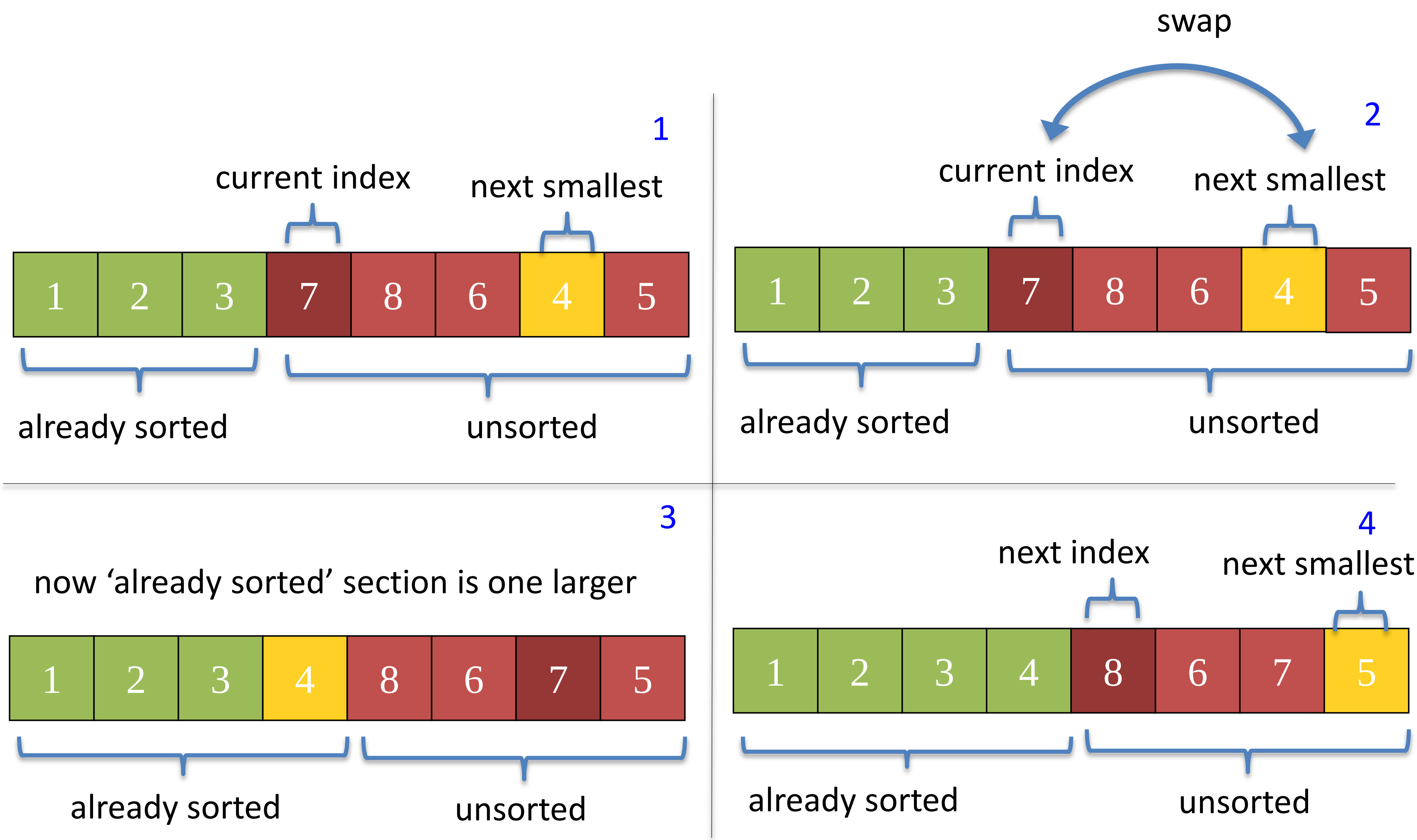
- Runtime?

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

  start sorted start reverse sorted (see text)

- Stable? Depends on implementation. Usually. In-place? Yes

Selection Sort



Selection Sort

- Idea: At step **`k`**, find the smallest element among the not-yet-sorted elements and put it at position k

```
for (int i = 0; i < n; i++) {
    // Find next smallest
    int newIndex = findNextMin(i);
    // Swap current and next smallest
    swap(newIndex, i);
}
```

- **Loop invariant**: when loop index is **`i`**, first **`i`** elements are sorted and **`i`** smallest elements in the array

- Runtime?
  Best-case _____ Worst-case _____ Average-case ____

- Stable? _____ In-place? _____

Selection Sort

- Idea: At step **`k`**, find the smallest element among the not-yet-sorted elements and put it at position k

```
for (int i = 0; i < n; i++) {
    // Find next smallest
    int newIndex = findNextMin(i);
    // Swap current and next smallest
    swap(newIndex, i);
}
```

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

- Runtime?
  Best-case, Worst-case, and Average-case $O(n^2)$

- Stable? Depends on implementation. Usually. In-place? Yes

Insertion Sort vs. Selection Sort

- Have the same worst-case and average-case asymptotic complexity
  - Insertion-sort has better best-case complexity; preferable when input is “mostly sorted”
- Useful for small arrays or for mostly sorted input

Bubble Sort

- for n iterations: ‘bubble’ next largest element to the end of the unsorted section, by doing a series of swaps

- Not intuitive – It’s unlikely that you’d come up with bubble sort

- Not good asymptotic complexity: $O(n^2)$

- It’s not particularly efficient with respect to common factors

Basically, almost never is better than insertion or selection sort.

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

Divide and conquer

Divide-and-conquer is a useful technique for solving many kinds of problems (not just sorting). It consists of the following steps:

1. Divide your work up into smaller pieces (recursively)
2. Conquer the individual pieces (as base cases)
3. Combine the results together (recursively)

```
algorithm(input) {
    if (small enough) {
        CONQUER, solve, and return input
    } else {
        DIVIDE input into multiple pieces
        RECURSE on each piece
        COMBINE and return results
    }
}
```

Divide-and-Conquer Sorting

Two great sorting methods are fundamentally divide-and-conquer

Mergesort:

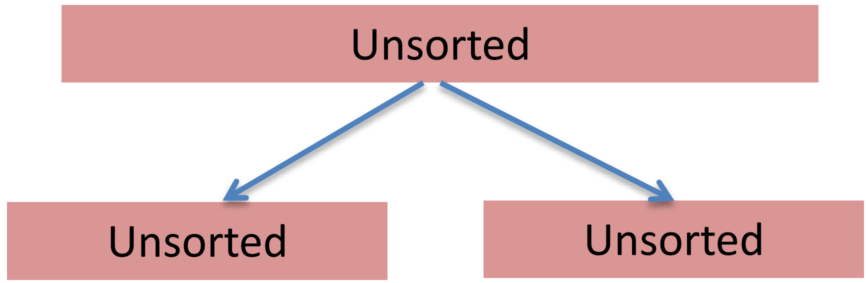
Sort the left half of the elements (recursively)
Sort the right half of the elements (recursively)
Merge the two sorted halves into a sorted whole

Quicksort:

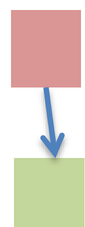
Pick a “pivot” element
Divide elements into less-than pivot and greater-than pivot
Sort the two divisions (recursively on each)
Answer is: sorted-less-than....pivot....sorted-greater-than

Merge Sort

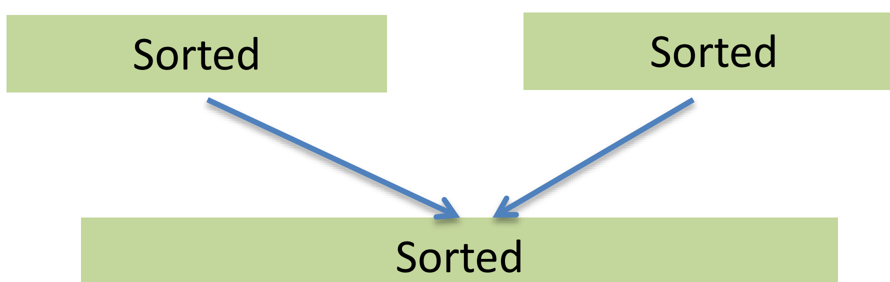
Divide: Split array roughly into half



Conquer: Return array when length ≤ 1



Combine: Combine two sorted arrays using merge

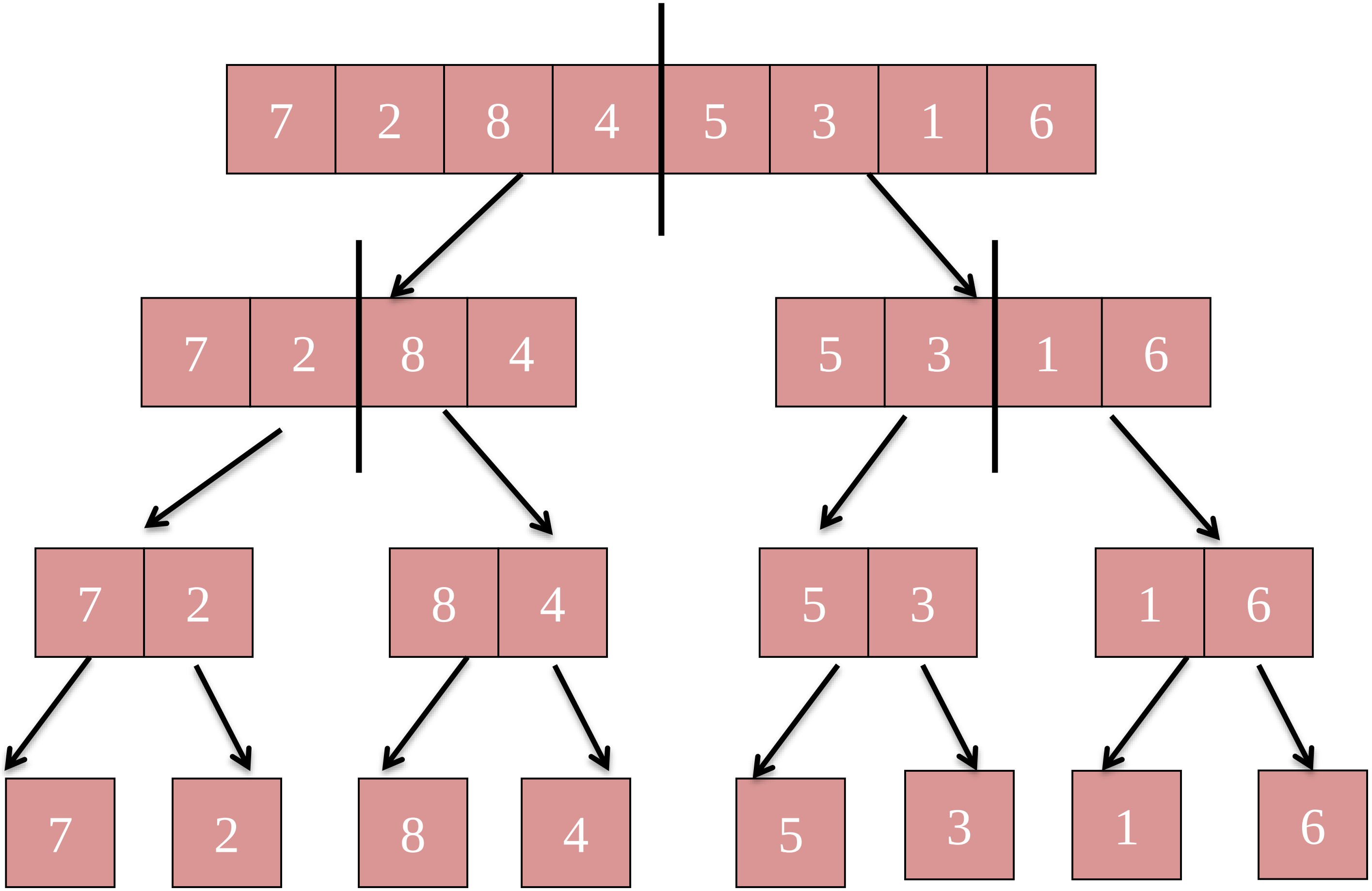


Merge Sort: Pseudocode

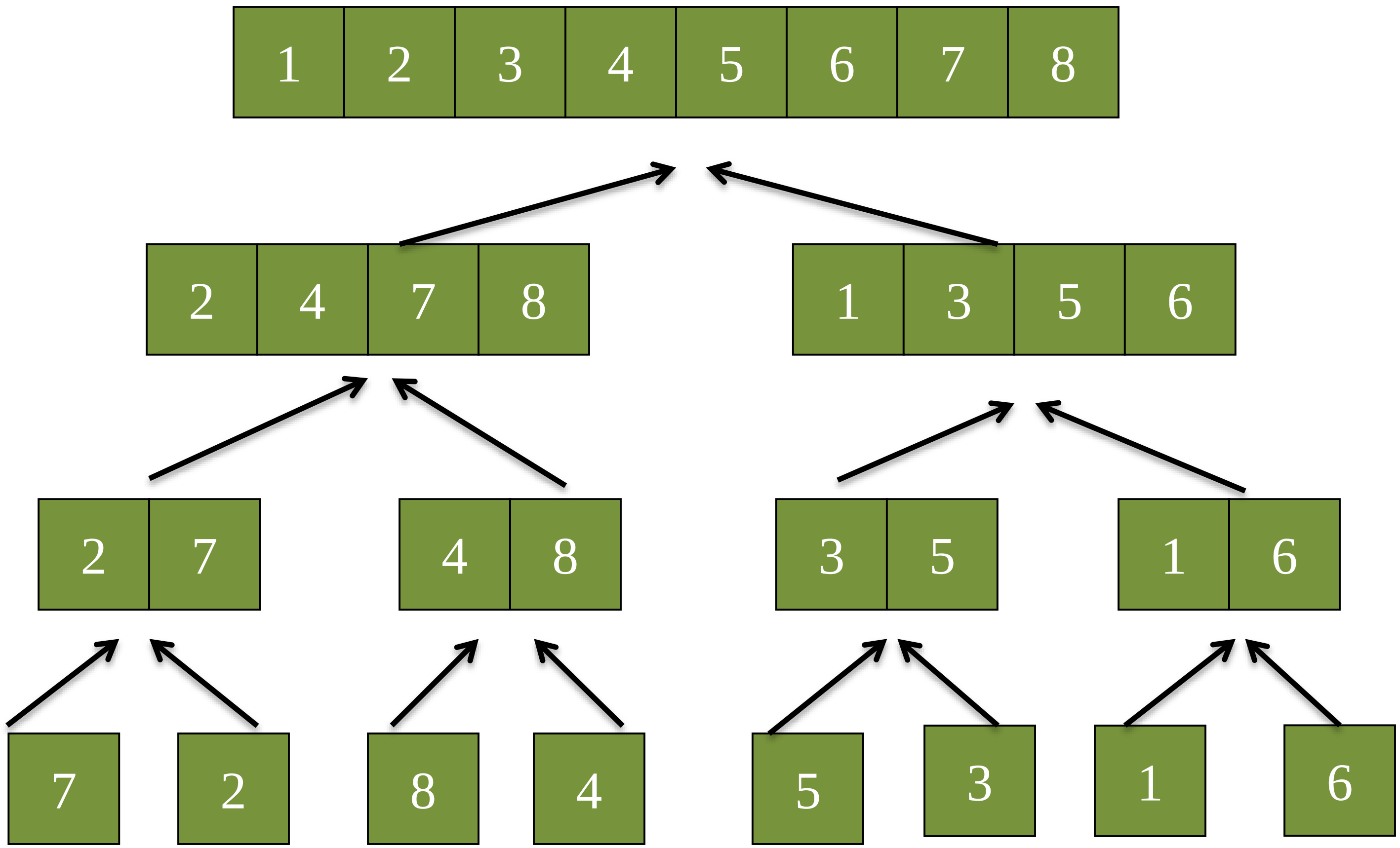
Core idea: split array in half, sort each half, merge back together. If the array has size 0 or 1, just return it unchanged

```
mergesort(input) {
    if (input.length < 2) {
        return input;
    } else {
        smallerHalf = sort(input[0, ..., mid]);
        largerHalf = sort(input[mid + 1, ...]);
        return merge(smallerHalf, largerHalf);
    }
}
```

Merge Sort Example

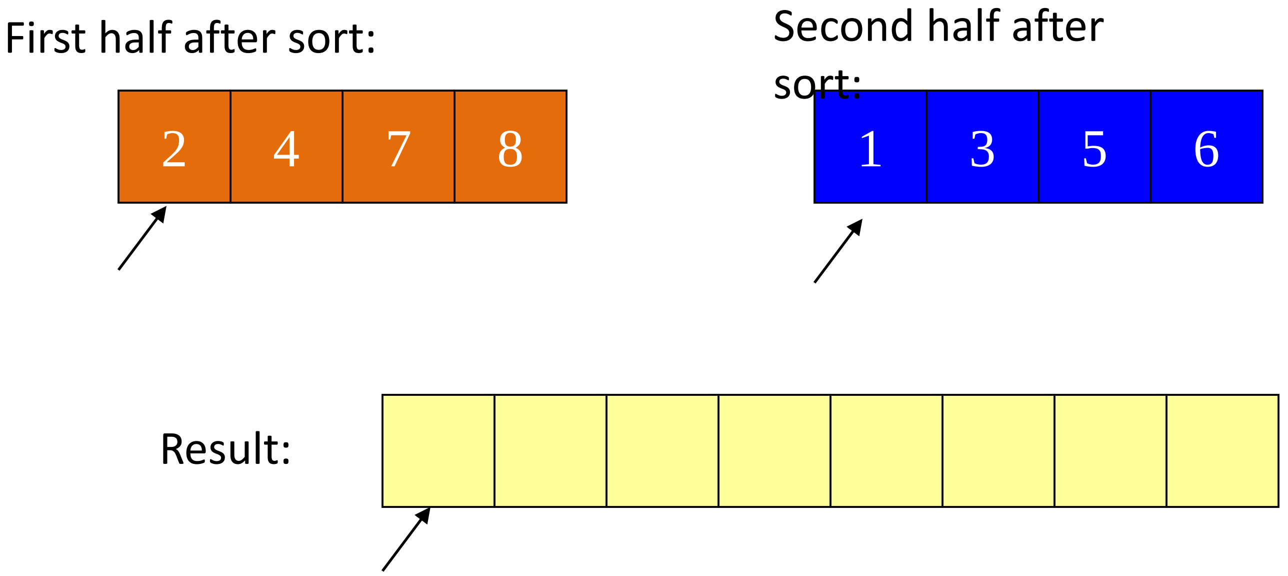


Merge Sort Example



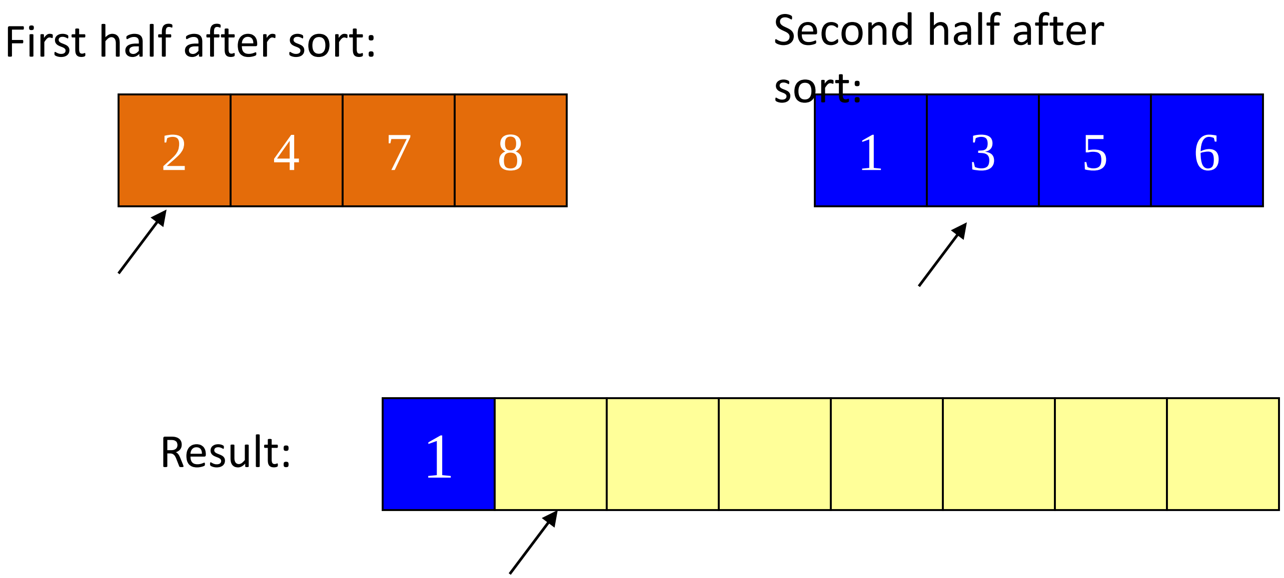
Merge Example

Merge operation: Use 3 pointers and 1 more array



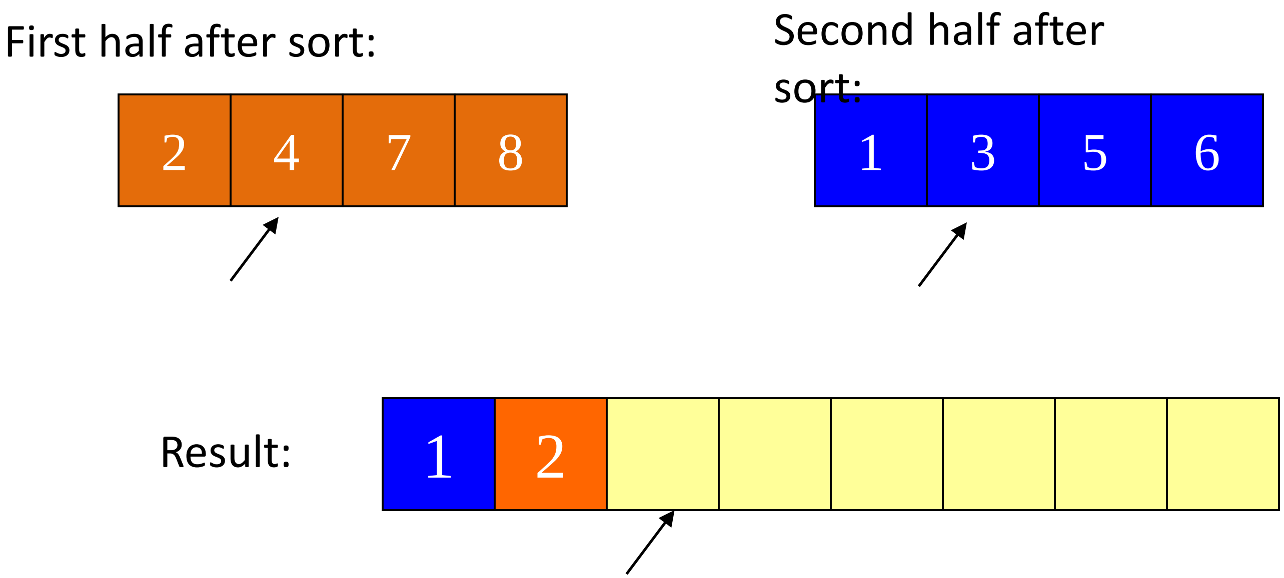
Merge Example

Merge operation: Use 3 pointers and 1 more array



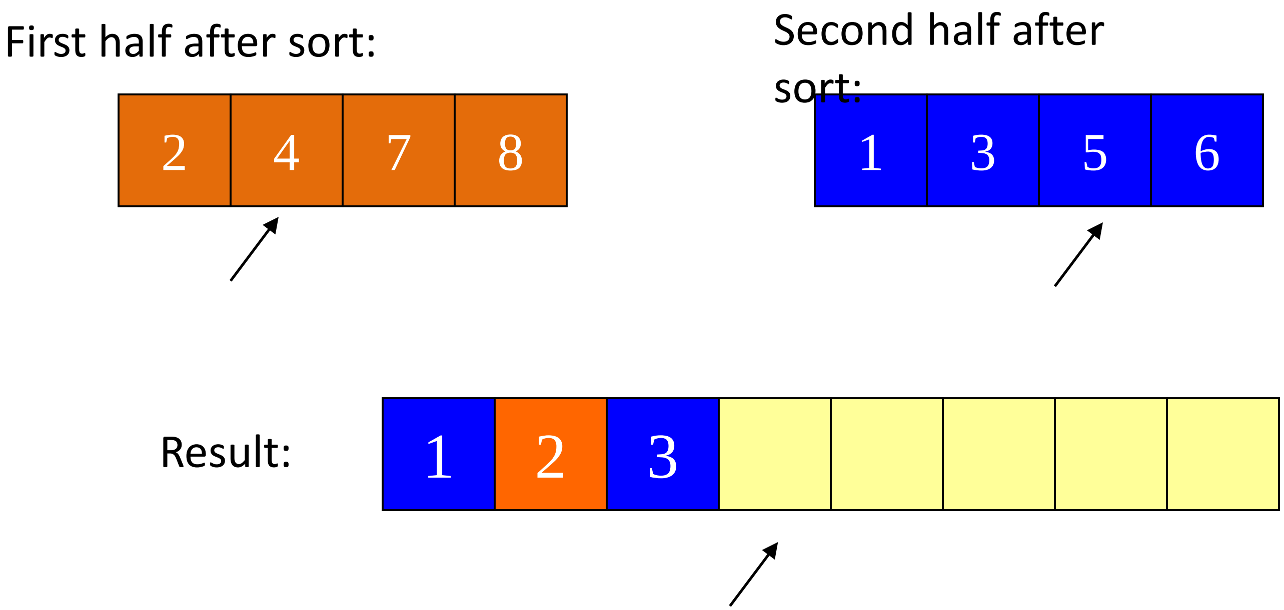
Merge Example

Merge operation: Use 3 pointers and 1 more array



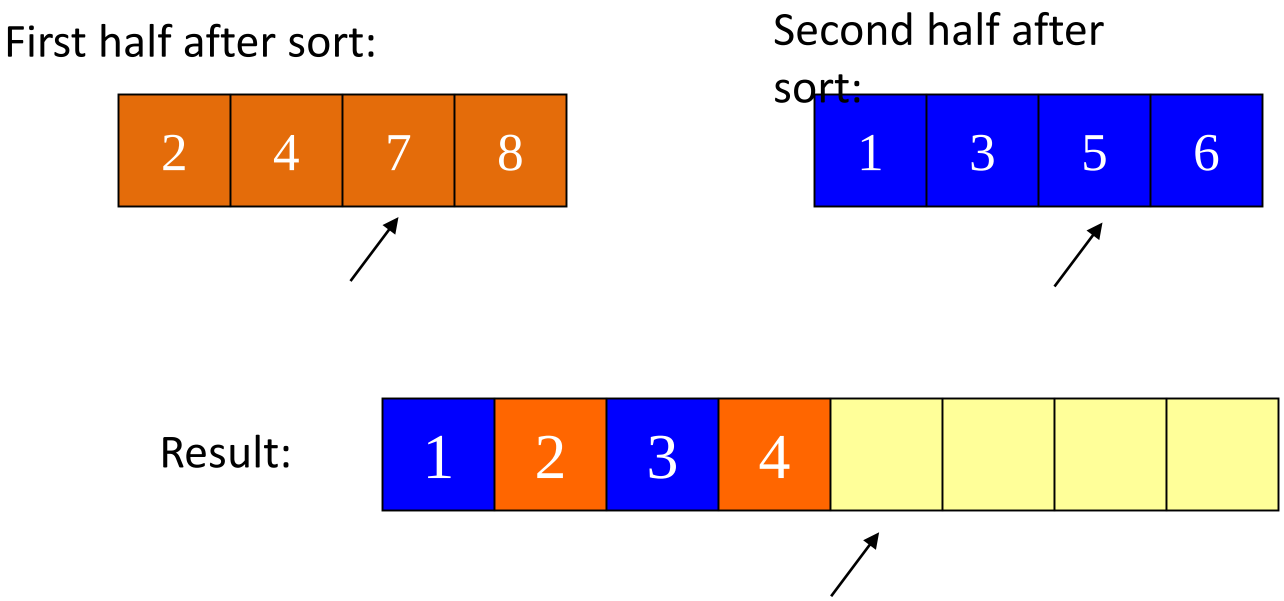
Merge Example

Merge operation: Use 3 pointers and 1 more array



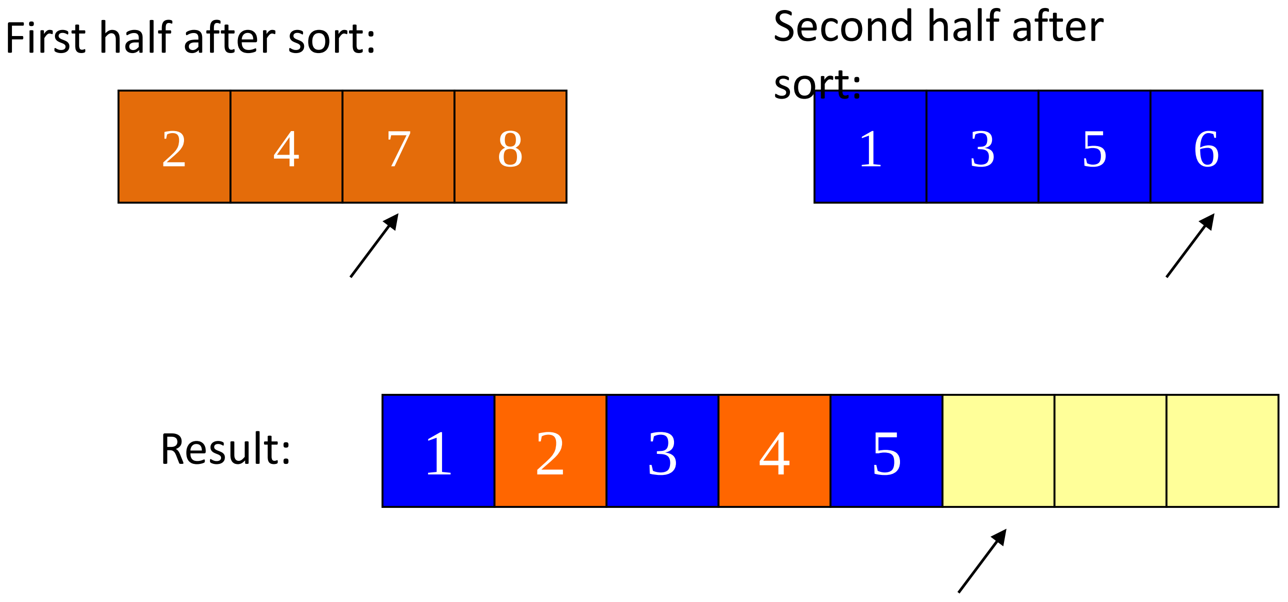
Merge Example

Merge operation: Use 3 pointers and 1 more array



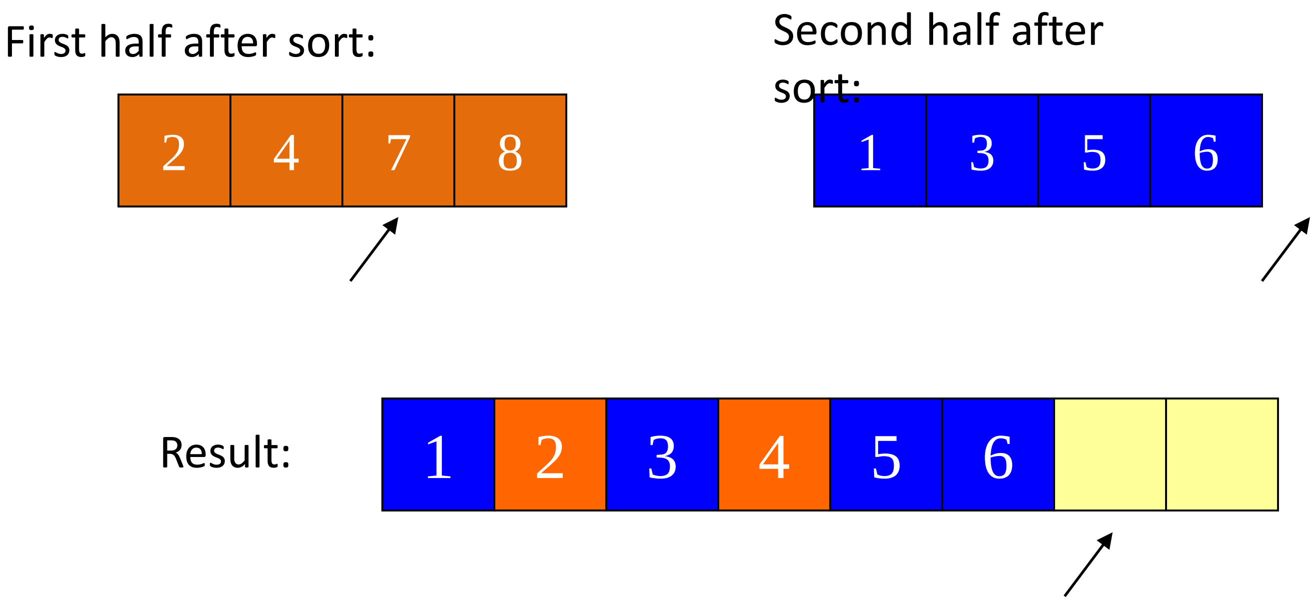
Merge Example

Merge operation: Use 3 pointers and 1 more array



Merge Example

Merge operation: Use 3 pointers and 1 more array



Merge Example

Merge operation: Use 3 pointers and 1 more array

First half after sort:

2	4	7	8

Second half after sort:

1	3	5	6

Result:

1	2	3	4	5	6	7	

Merge Example

Merge operation: Use 3 pointers and 1 more array

First half after sort:

2	4	7	8

Second half after sort:

1	3	5	6

Result:

1	2	3	4	5	6	7	8

After Merge: copy result into original unsorted array.

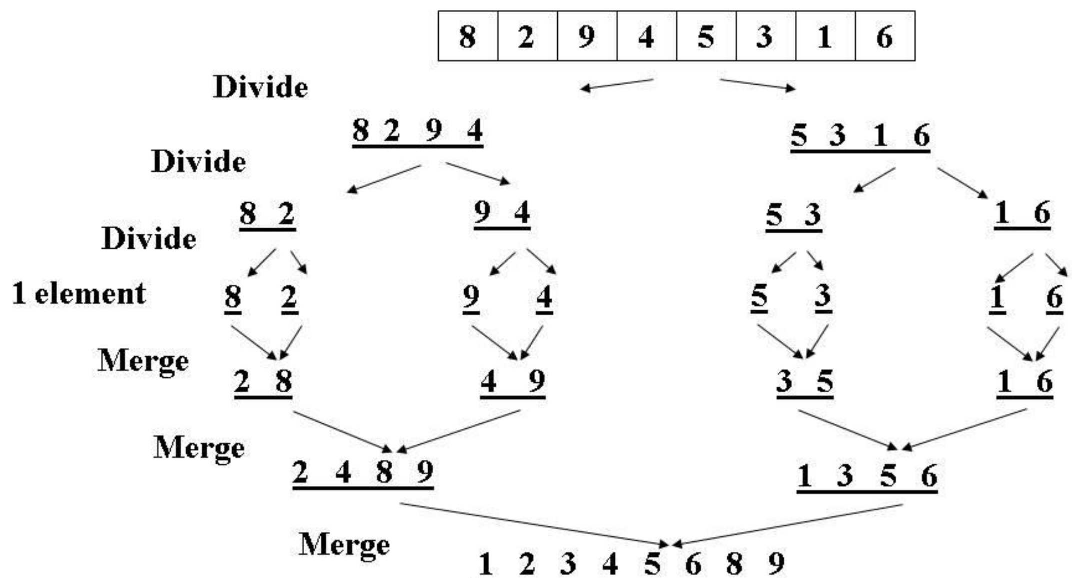
Or you can do the whole process in-place, but it's more difficult to write

Merge Sort Analysis

Runtime:

- subdivide the array in half each time: O(log(n)) recursive calls
- merge is an O(n) traversal at each level

So, the best and worst case runtime is the same: O(n log(n))



O(log(n)) levels

Merge Sort Analysis

Stable?

Yes! If we implement the merge function correctly, merge sort will be stable.

In-place?

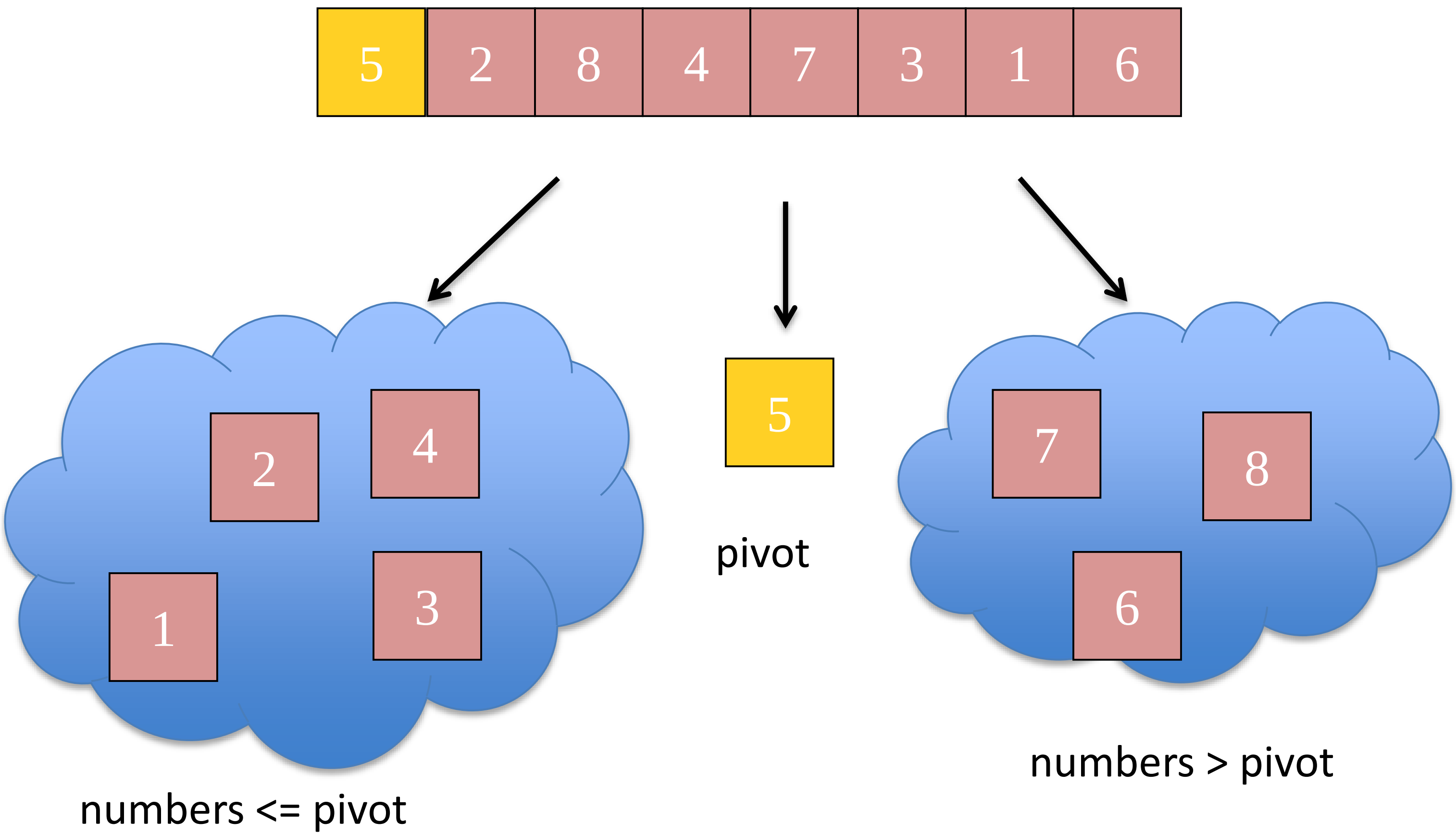
No. Unless you want to give yourself a headache. Merge must construct a new array to contain the output, so merge sort is not in-place.

We're constantly copying and creating new arrays at each level...

One Solution: (less of a headache than actually implementing in-place) create a single auxiliary array and swap between
it and the original on each level.

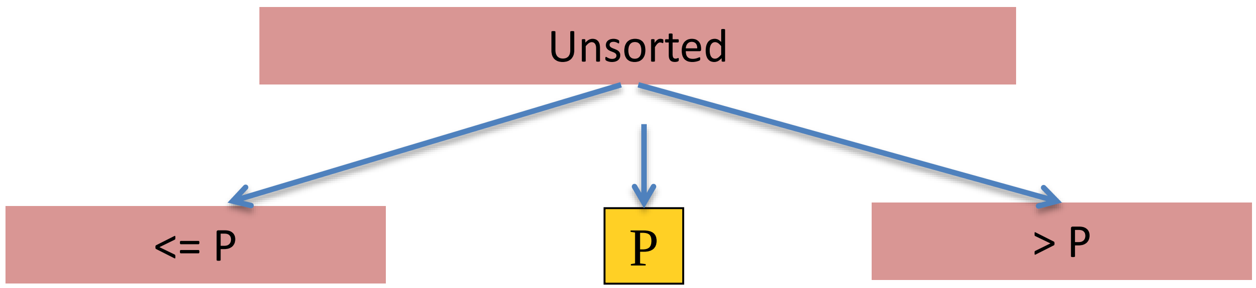
Quick Sort

Divide: Split array around a 'pivot'

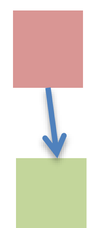


Quick Sort

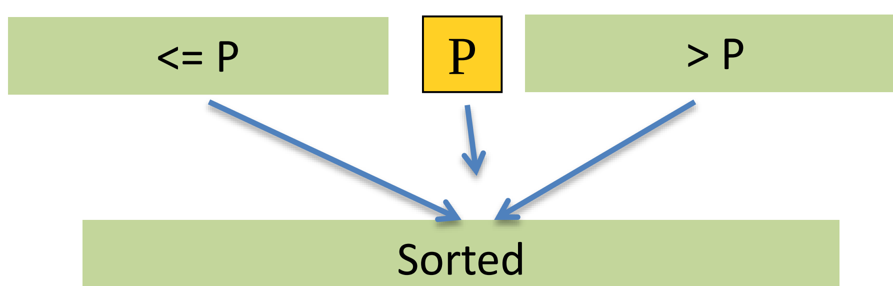
Divide: Pick a pivot, partition into groups



Conquer: Return array when length ≤ 1



Combine: Combine sorted partitions and pivot

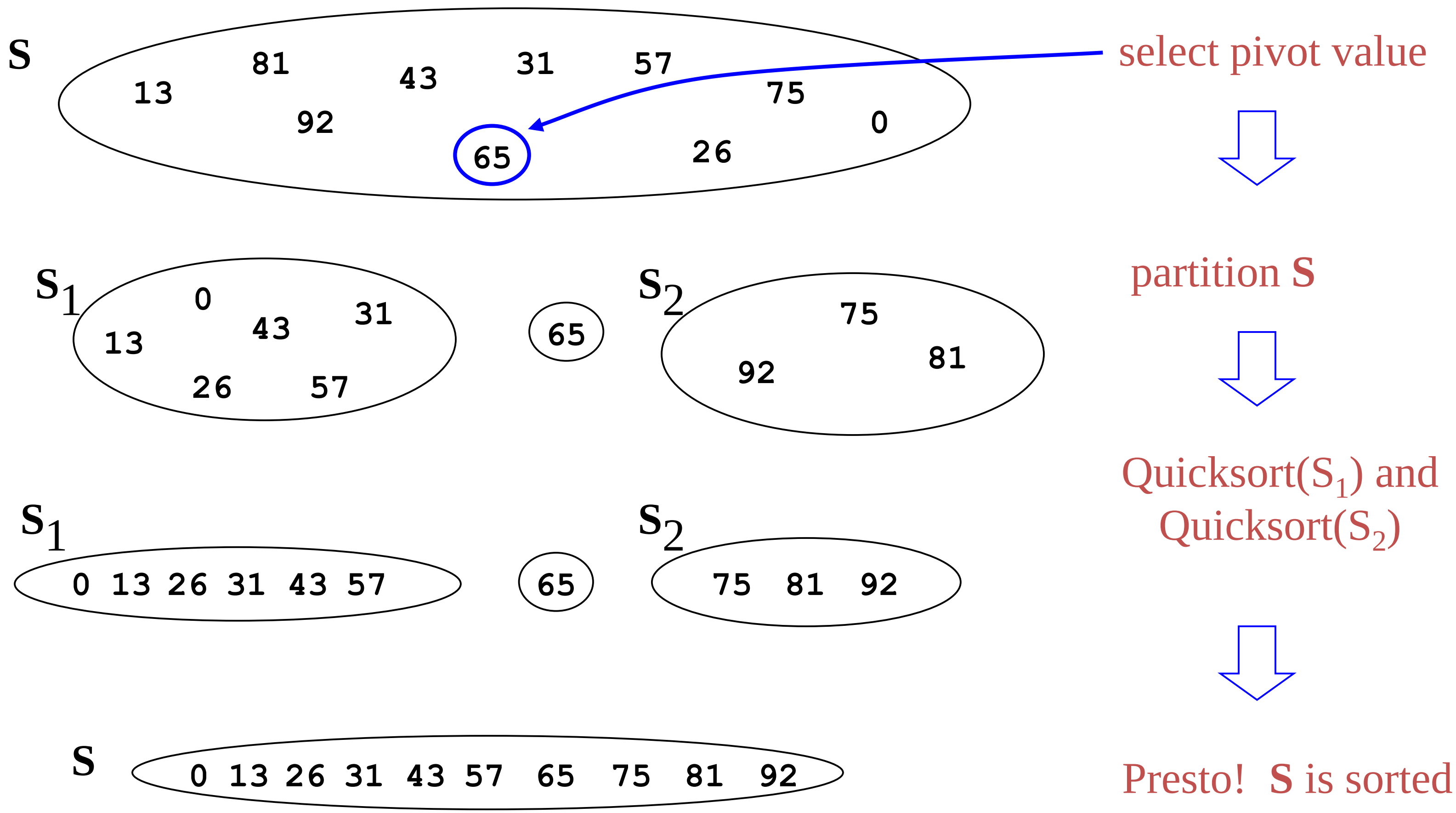


Quick Sort Pseudocode

Core idea: Pick some item from the array and call it the pivot. Put all items smaller in the pivot into one group and all items larger in the other and recursively sort. If the array has size 0 or 1, just return it unchanged.

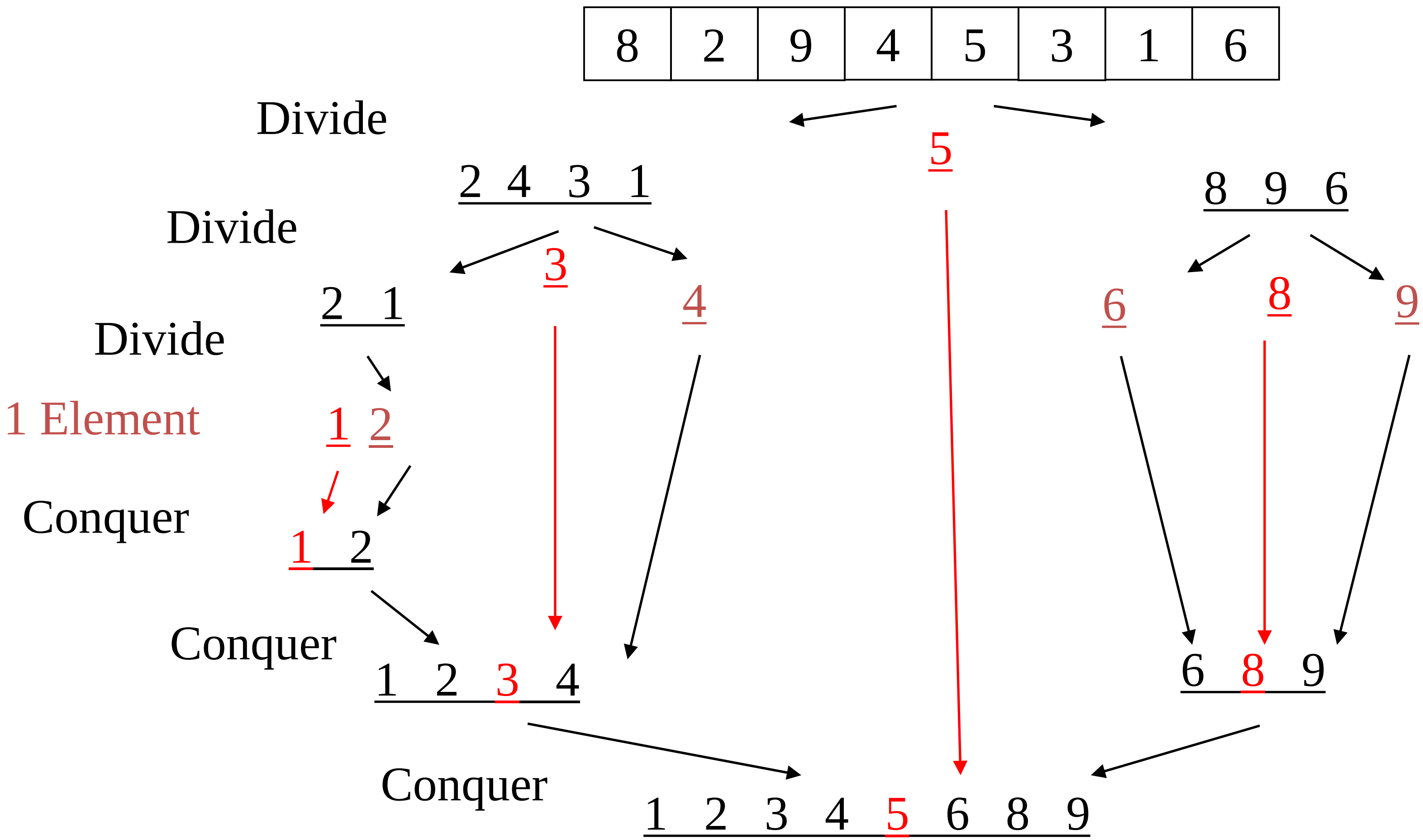
```
quicksort(input) {
    if (input.length < 2) {
        return input;
    } else {
        pivot = getPivot(input);
        smallerHalf = sort(getSmaller(pivot, input));
        largerHalf = sort(getBigger(pivot, input));
        return smallerHalf + pivot + largerHalf;
    }
}
```

Think in Terms of Sets



[Weiss]

Example, Showing Recursion



Details

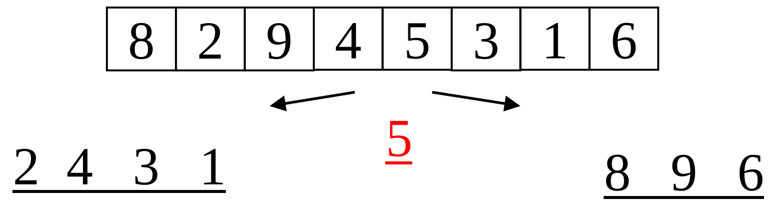
Have not yet explained:

- 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

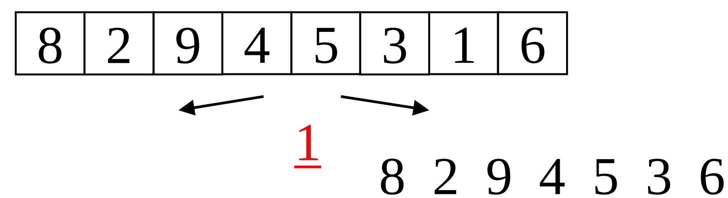
- How to implement partitioning
  - In linear time
  - In place

Pivots

- Best pivot?
  - Median
  - Halve each time



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



Potential pivot rules

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

- 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]`** *

*skip step 4 if pivot ends up being least element

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
  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
  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)
  - $T(n) = n + \frac{(n-1)!}{n!}\left[\sum_{i=1}^{n} T(i-1) + T(n-i)\right]$
  - $O(n \log n)$, not responsible for proof (in text)

Quick Sort Analysis

- **In-place:** Yep! We can use a couple pointers and partition the array in place, recursing on different **`lo`** and **`hi`** `indices`
- **Stable**: Not necessarily. Depends on how you handle equal values when partitioning. A stable version of quick sort uses some extra storage for partitioning.

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

How Fast Can We Sort?

- (Heapsort &) Mergesort have $O(n \texttt{ log } n)$ worst-case running time

- Quicksort has $O(n \texttt{ log } n)$ average-case running time

- These bounds are all tight, actually $\Theta(n \texttt{ log } n)$

- ***Assuming* our comparison *model***: The only operation an algorithm can perform on data items is a 2-element comparison. There is no lower asymptotic complexity, such as $O(n)$ or $O(n \texttt{ log log } n)$

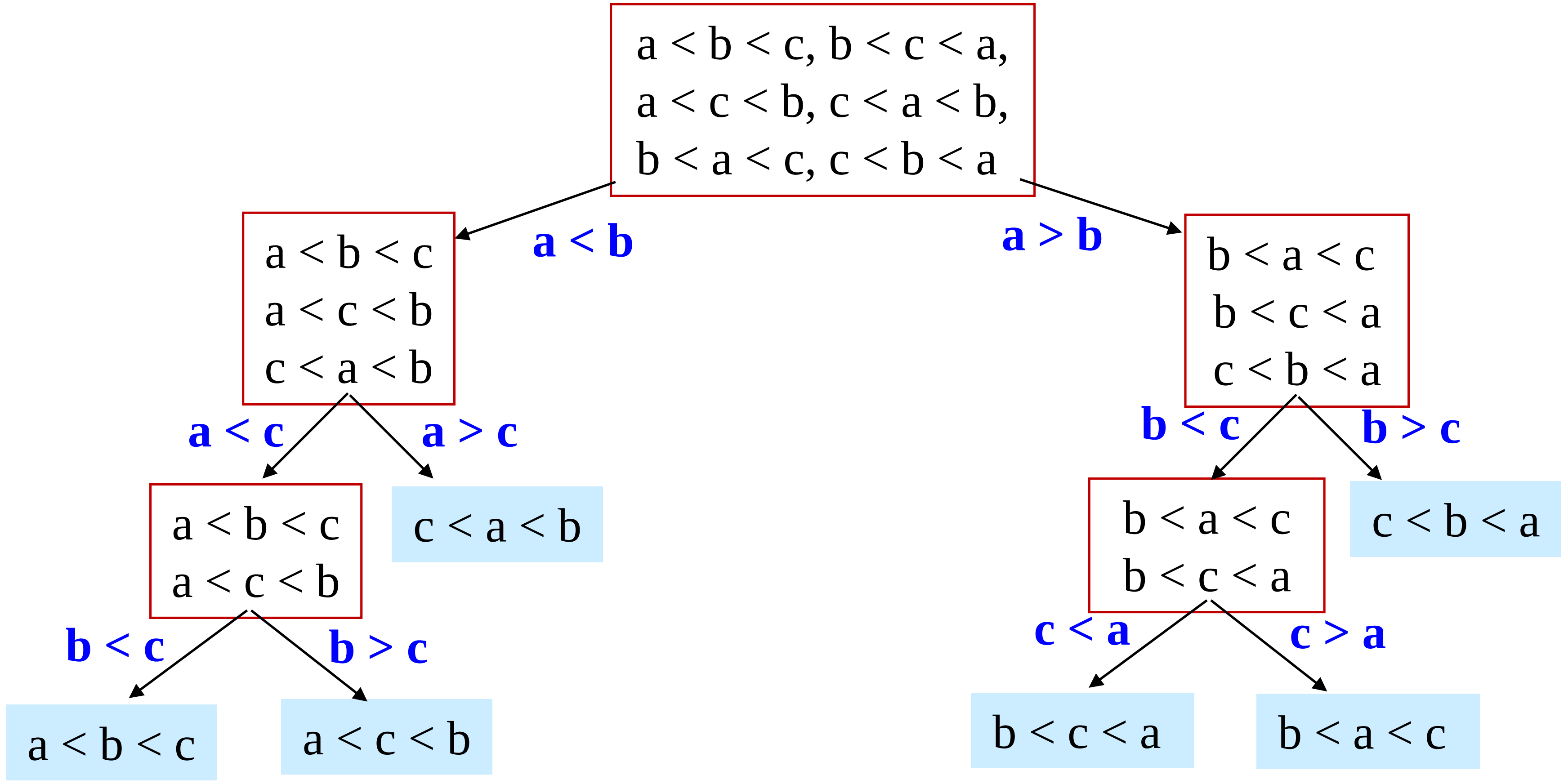
Counting Comparisons

- No matter what the algorithm is, it cannot make progress without doing comparisons

- **Intuition**: Each comparison can *at best* eliminate *half* the remaining possibilities of possible orderings

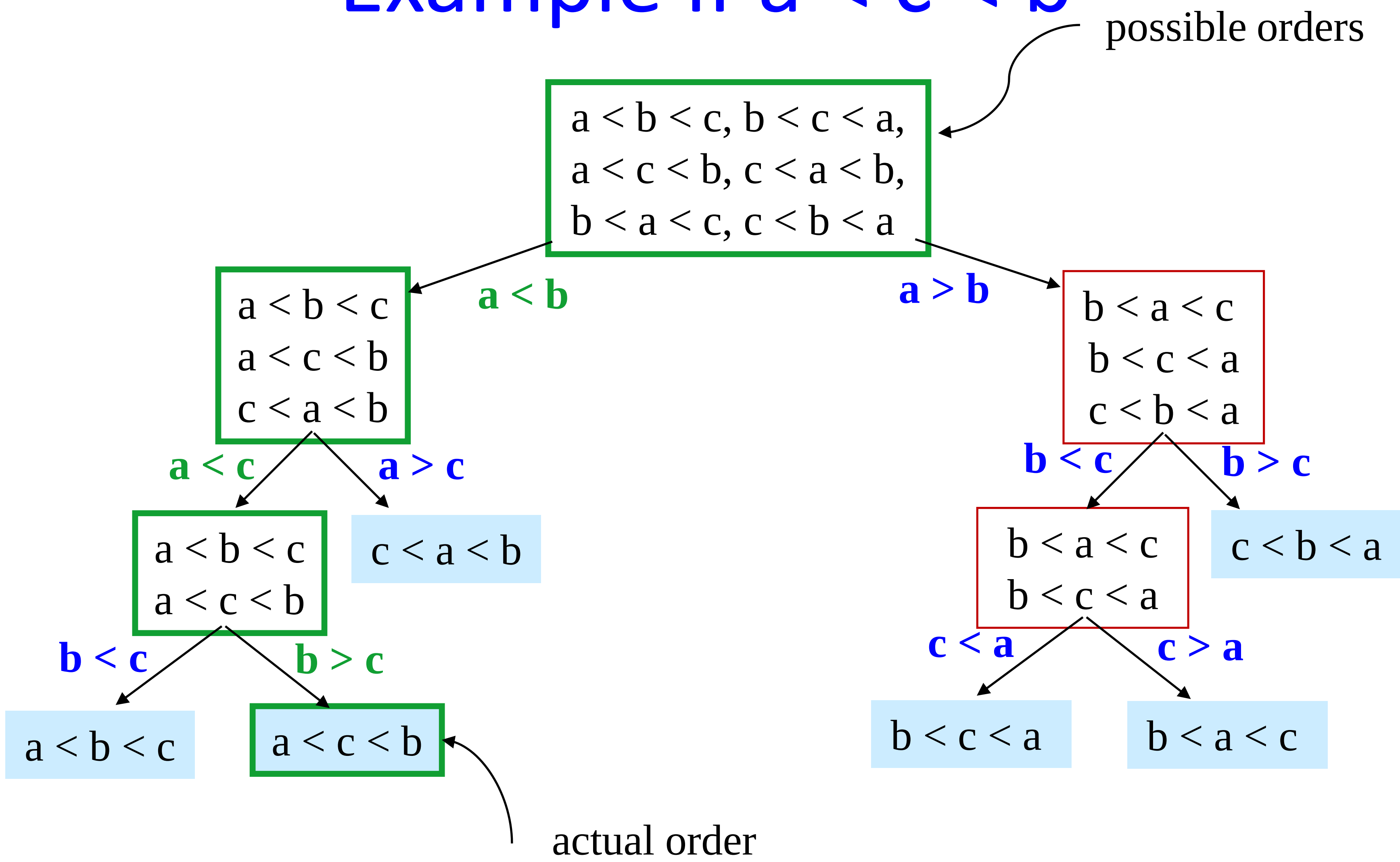
- Can represent this process as a *decision tree*
  - Nodes contain "set of remaining possibilities"
  - Edges are "answers from a comparison"
  - The algorithm does not actually build the tree; it's what our *proof* uses to represent "the most the algorithm could know so far" as the algorithm progresses

Decision Tree for n = 3



- The leaves contain all the possible orderings of a, b, c

Example if a < c < b



What the Decision Tree Tells Us

- A binary tree because each comparison has 2 outcomes (we're comparing 2 elements at a time)
- Because any data is possible, any algorithm needs to ask enough questions to produce all orderings.

The facts we can get from that:

1. Each ordering is a different leaf (only one is correct)
2. Running *any* algorithm on *any* input will *at best* correspond to a root-to-leaf path in *some* decision tree. Worst number of comparisons is the longest path from root-to-leaf in the decision tree for input size n
3. There is no worst-case running time better than the height of a tree with *<num possible orderings>* leaves

How many possible orderings?

- Assume we have *n* elements to sort. How many *permutations* of the elements (possible orderings)?
  - For simplicity, assume none are equal (no duplicates)

Example, ***n*=3**

a[0]<a[1]<a[2]	a[0]<a[2]<a[1]	a[1]<a[0]<a[2]
a[1]<a[2]<a[0]	a[2]<a[0]<a[1]	a[2]<a[1]<a[0]

In general, *n* choices for least element, *n*-1 for next, *n*-2 for next, …

- $n(n\text{-}1)(n\text{-}2)\ldots(2)(1) = \boldsymbol{n!}$ possible orderings

That means with n! possible leaves, **best height for tree is log(n!)**, given that **best case tree** splits leaves in half at each branch

What does that mean for runtime?

That proves runtime is at least Ω(**`log`** (*n!)*). Can we write that more clearly?

$$
\begin{aligned}
\lg(n!) &= \lg(n(n-1)(n-2)\ldots 1) && \text{[Def. of } n!\text{]}\\
&= \lg(n) + \lg(n-1) + \ldots \lg\left(\frac{n}{2}\right) + \lg\left(\frac{n}{2}-1\right) + \ldots \lg(1) && \text{[Prop. of Logs]}\\
&\geq \lg(n) + \lg(n-1) + \ldots + \lg\left(\frac{n}{2}\right)\\
&\geq \left(\frac{n}{2}\right)\lg\left(\frac{n}{2}\right)\\
&= \left(\frac{n}{2}\right)(\lg n - \lg 2)\\
&= \frac{n \lg n}{2} - \frac{n}{2}\\
&\in \Omega(n \lg(n))
\end{aligned}
$$

Nice! Any sorting algorithm must do *at best* (1/2)*(*n* **`log`** *n* **`-`** *n)* comparisons: Ω(*n* **`log`** *n*)

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

BucketSort (a.k.a. BinSort)

- If all values to be sorted are known to be integers between 1 and *K* (or any small range):
  - Create an array of size *K*
  - Put each element in its proper bucket (a.k.a. bin)
  - *If* data is only integers, no need to store more than a *count* of how times that bucket has been used
- Output result via linear pass through array of buckets

`count` array	
1	3
2	1
3	2
4	2
5	3

- Example:

  K=5

  input (5,1,3,4,3,2,1,1,5,4,5)

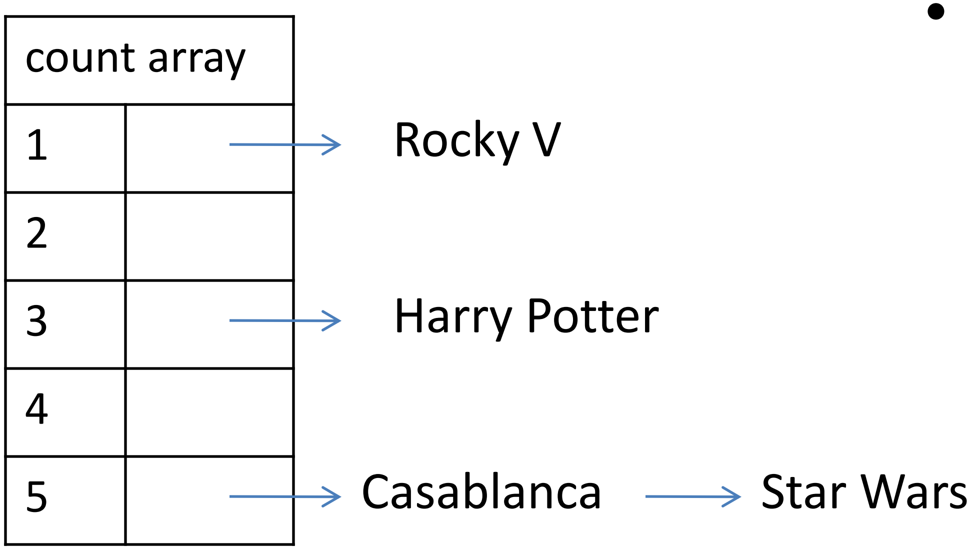
  output: 1,1,1,2,3,3,4,4,5,5,5

Analyzing Bucket Sort

- **Overall: $O(n+K)$**
  - Linear in n, but also linear in K
- Good when K is smaller (or not much larger) than n
  - We don't spend time doing comparisons of duplicates
- Bad when K is much larger than n
  - Wasted space; wasted time during linear $O(K)$ pass
- For data in addition to integer keys, use list at each bucket

Bucket Sort with non integers

- Most real lists aren't just keys; we have data
- Each bucket is a list (say, linked list)
- To add to a bucket, insert in $O(1)$ (at beginning, or keep pointer to last element)



- Example: Movie ratings; scale 1-5

  Input:

  5: Casablanca

  3: Harry Potter movies

  5: Star Wars Original Trilogy

  1: Rocky V

- Result: 1: Rocky V, 3: Harry Potter, 5: Casablanca, 5: Star Wars
- Easy to keep 'stable'; Casablanca still before Star Wars

Radix sort

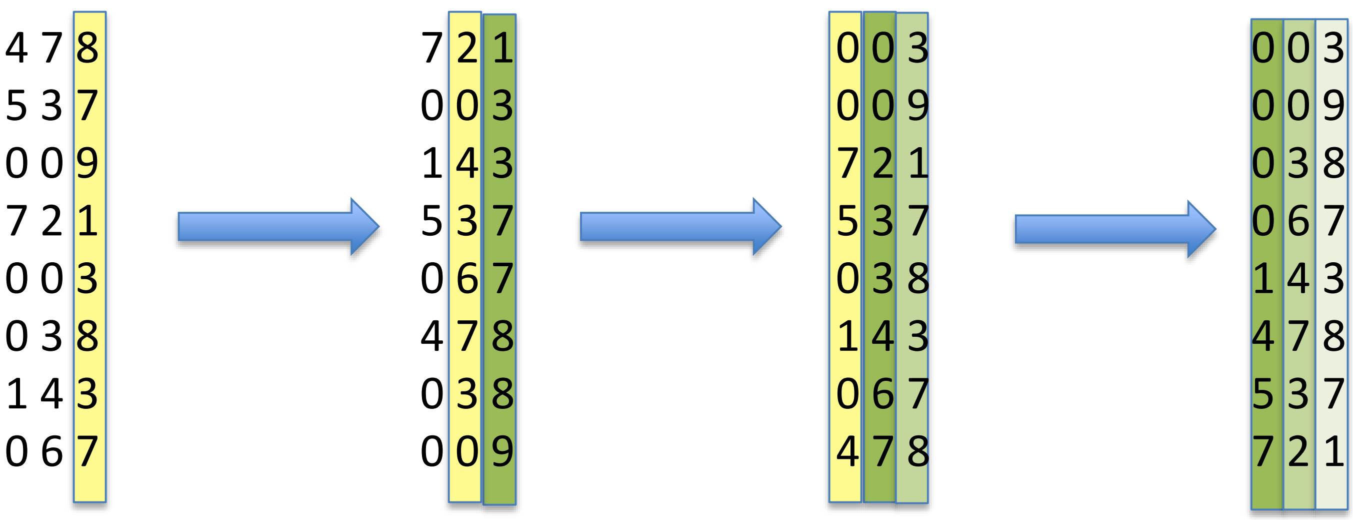
- Radix = “the base of a number system”
  - Examples will use base 10 because we are used to that
  - In implementations use larger numbers
    - For example, for ASCII strings, might use 128
- **Idea**:
  - Bucket sort on one digit at a time
    - Number of buckets = radix
    - Starting with *least* significant digit
    - Keeping sort *stable*
  - Do one pass per digit
  - **Invariant**: After *k* passes (digits), the last *k* digits are sorted

Radix Sort Example

Radix = 10

Input: 478, 537, 9, 721, 3, 38, 143, 67

3 passes (input is 3 digits at max), on each pass, stable sort the input highlighted in yellow



Example

Radix = 10

0	1	2	3	4	5	6	7	8	9
	721		3 143				537 67	478 38	9

Input: 478
537
9
721
3
38
143
67

First pass:

bucket sort by ones digit

Order now:

721
003
143
537
067
478
038
009

Example

Radix = 10

0	1	2	3	4	5	6	7	8	9
	721		3 143				537 67	478 38	9

0	1	2	3	4	5	6	7	8	9
3 9		721	537 38	143		67	478		

Order was:

721
003
143
537
067
478
038
009

Second pass:

stable bucket sort by tens digit

Order now:

003
009
721
537
038
143
067
478

Example

0	1	2	3	4	5	6	7	8	9
3 9		721	537 38	143		67	478		

Radix = 10

0	1	2	3	4	5	6	7	8	9
3 9 38 67	143			478	537		721		

Order was:

003
009
721
537
038
143
067
478

Third pass:

stable bucket sort by 100s digit

Order now:

003
009
038
067
143
478
537
721

Analysis

Input size: n

Number of buckets = Radix: B

Number of passes = "Digits": P

Work per pass is 1 bucket sort: $O(B+n)$

Total work is $\boldsymbol{O(P(B+n))}$

Compared to comparison sorts, sometimes a win, but often not

- Example: Strings of English letters up to length 15
  - Run-time proportional to: $15*(52 + n)$
  - This is less than n log n only if $n > 33{,}000$
  - Of course, cross-over point depends on constant factors of the implementations

Sorting Takeaways

- 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
- $\Omega(n \log n)$ 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!