CSE 373

NOVEMBER 13TH – MERGING AND PARTITIONING

REVIEW

- **Slow sorts**
  - $O(n^2)$
  - Insertion
  - Selection
- **Fast sorts**
  - $O(n \log n)$
  - Heap sort

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:

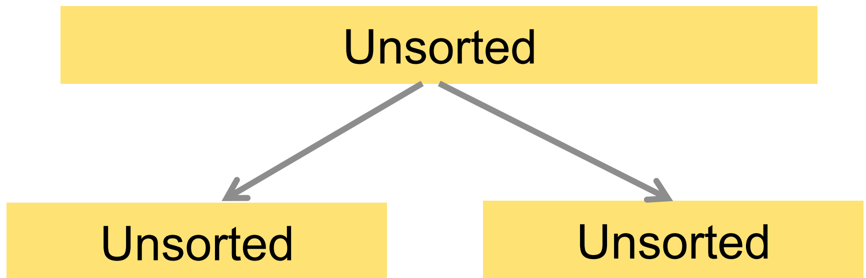
Sort the left half of the elements (recursively)
Sort the right half of the elements (recursively)
Merge the two sorted halves into a sorted whole

Quicksort:

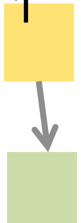
Pick a “pivot” element
Divide elements into less-than pivot and greater-than pivot
Sort the two divisions (recursively on each)
Answer is: sorted-less-than....pivot....sorted-greater-than

MERGE SORT

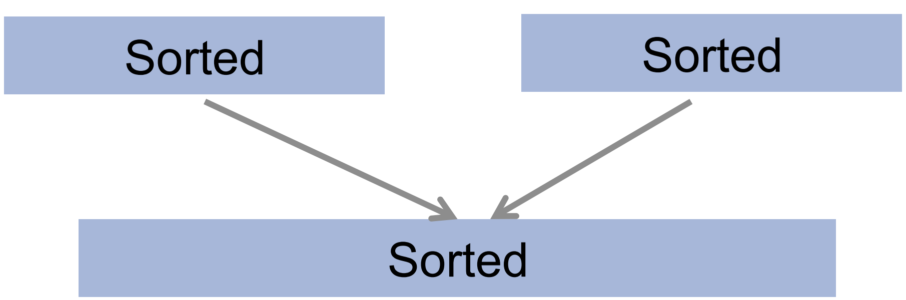
Divide: Split array roughly into half



Conquer: Return array when length ≤ 1



Combine: Combine two sorted arrays using merge

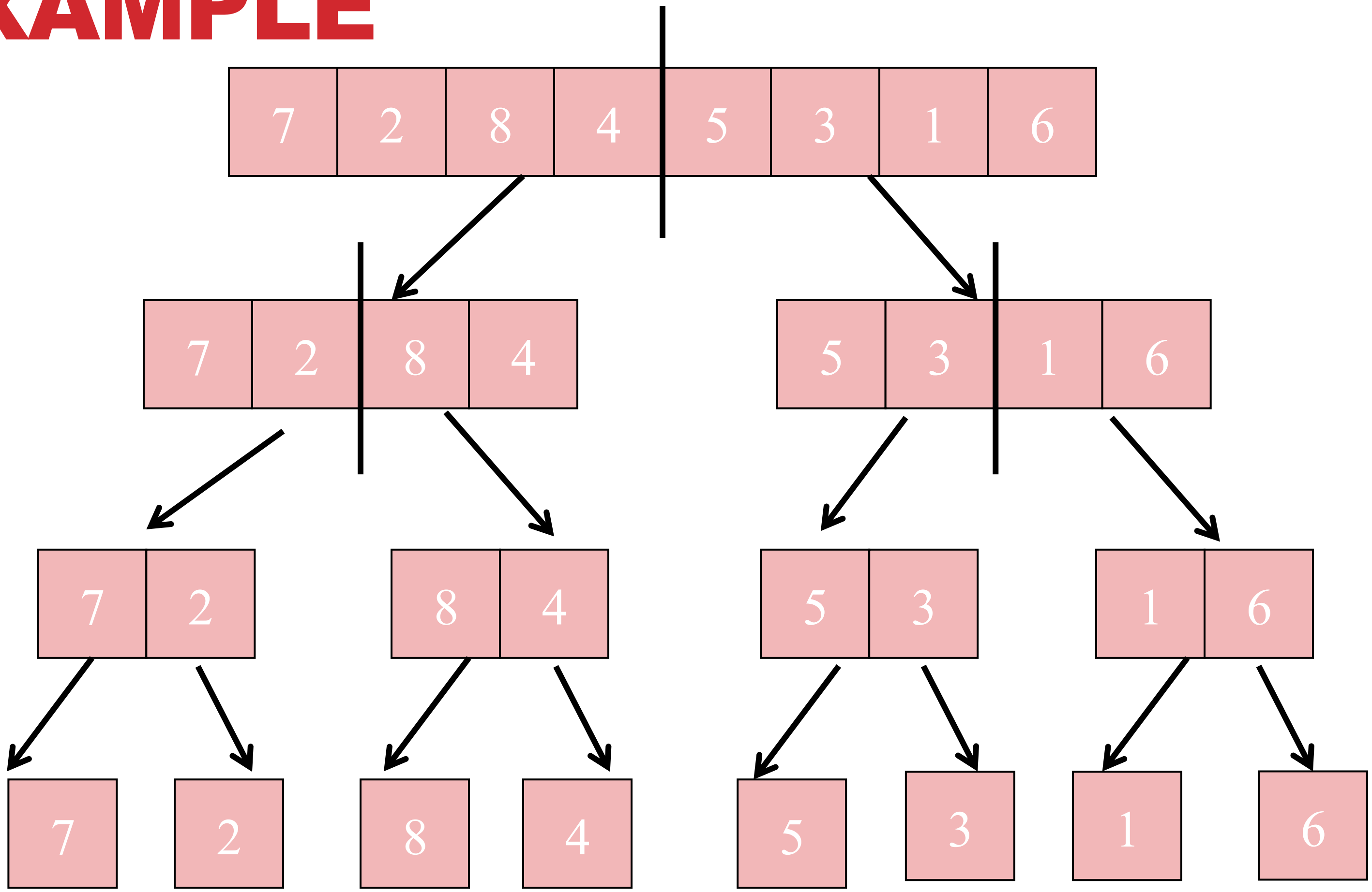


MERGE SORT: PSEUDOCODE

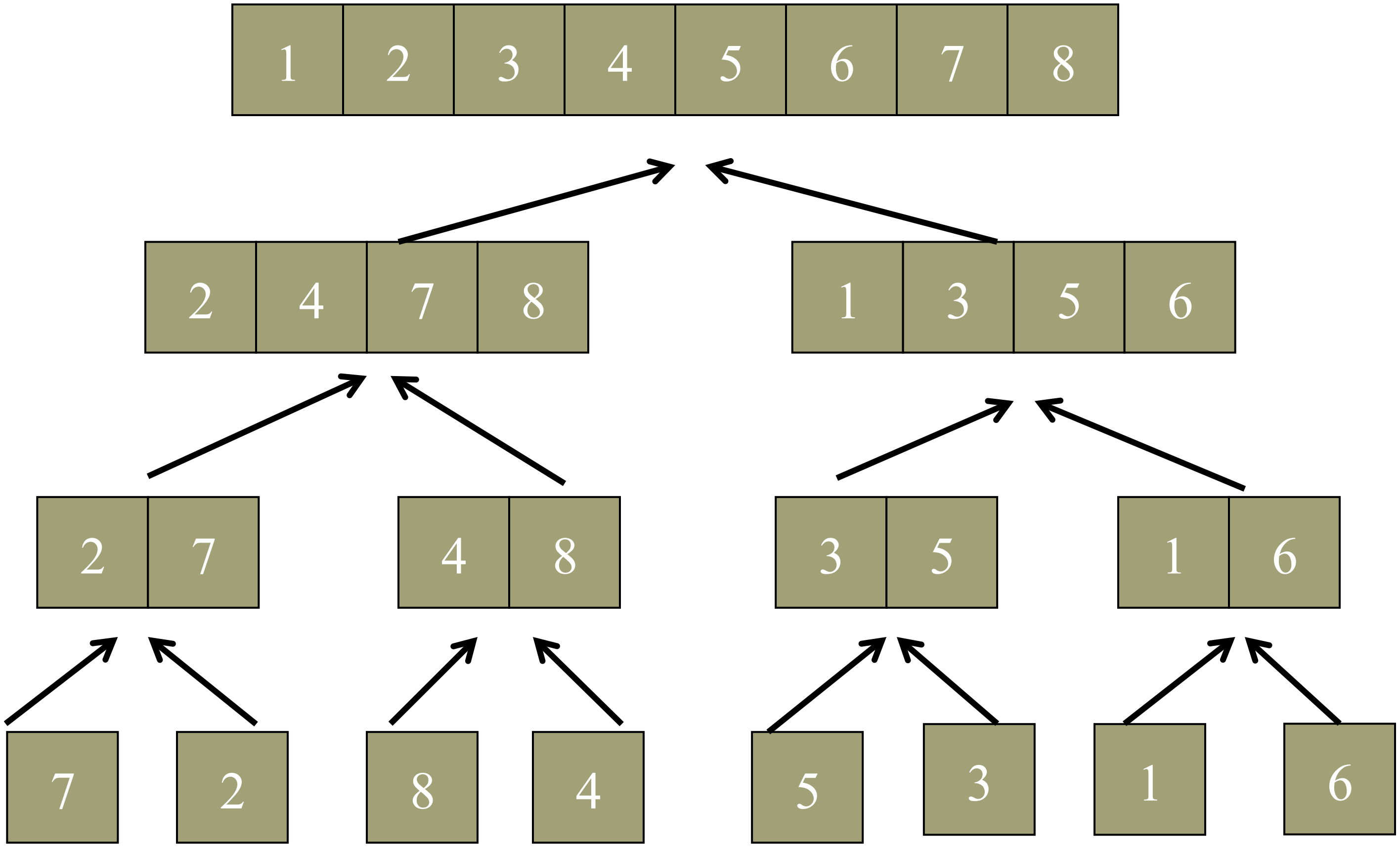
Core idea: split array in half, sort each half, merge back together. If the array has size 0 or 1, just return it unchanged

```
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



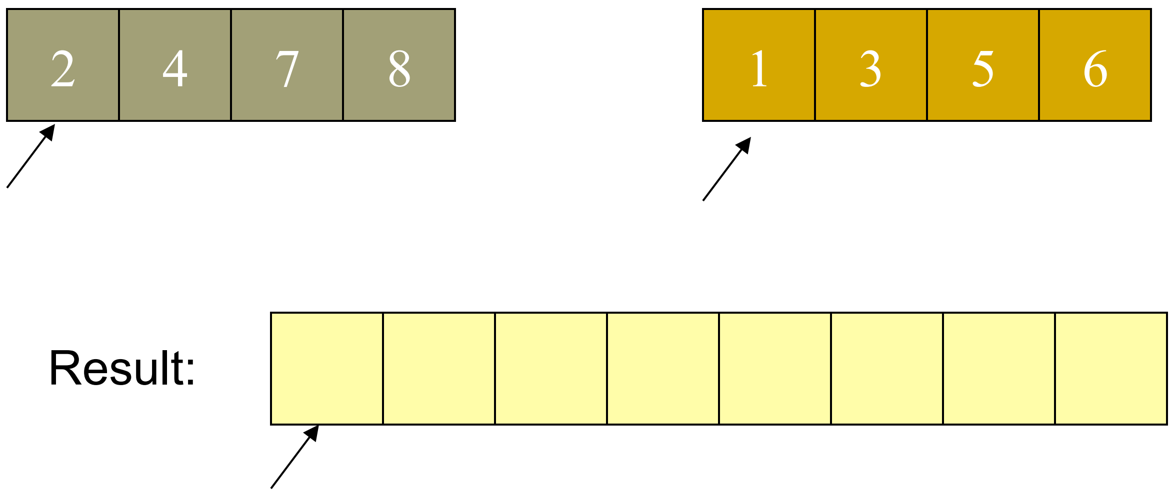
MERGE EXAMPLE

Merge operation: Use 3 pointers and 1 more array

First half after sort:

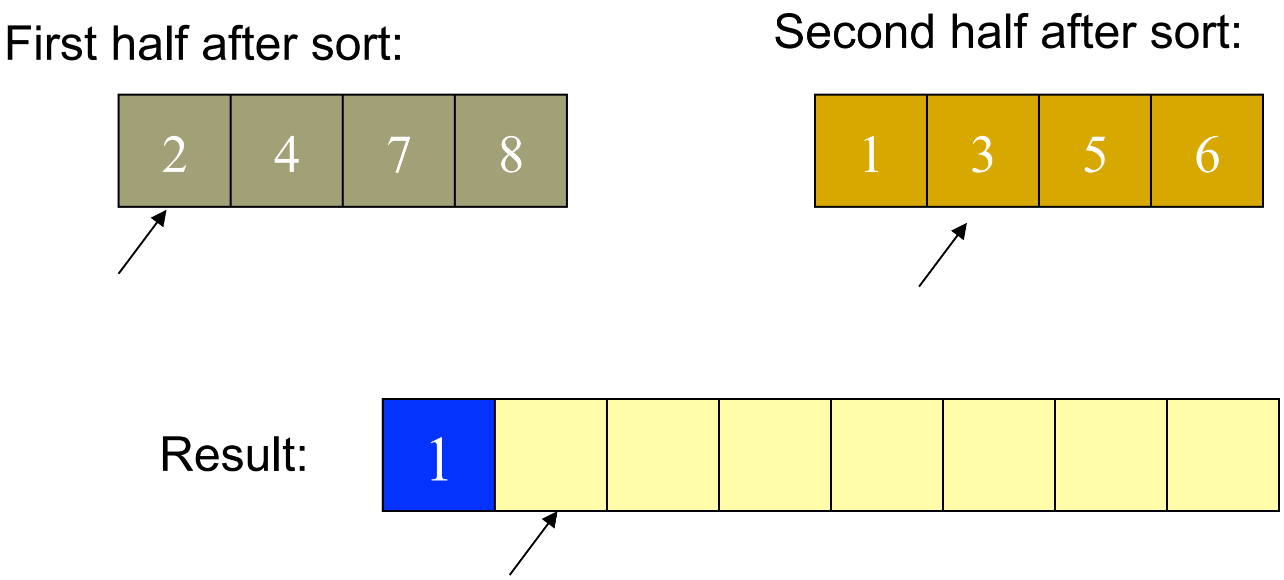
Second half after sort:

Result:



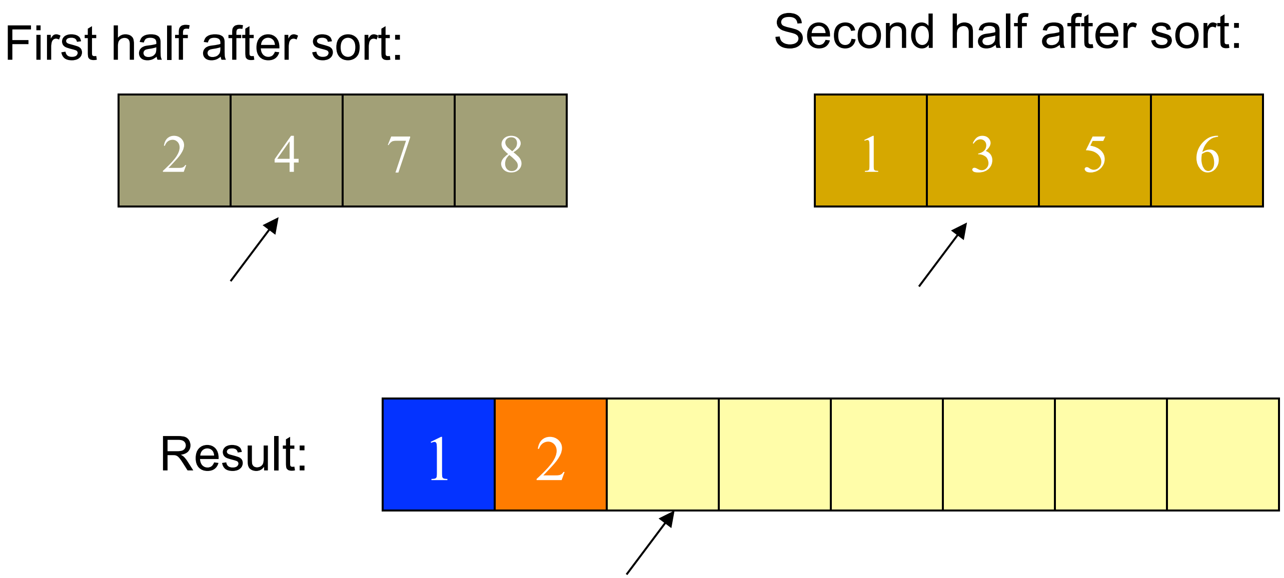
MERGE EXAMPLE

Merge operation: Use 3 pointers and 1 more array



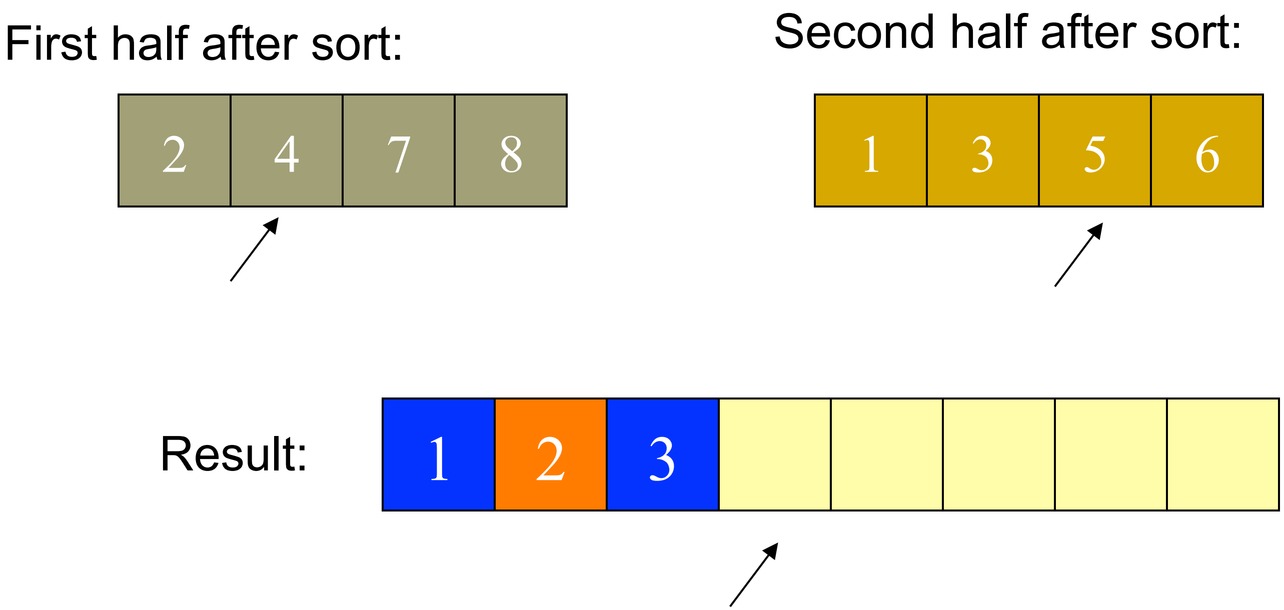
MERGE EXAMPLE

Merge operation: Use 3 pointers and 1 more array



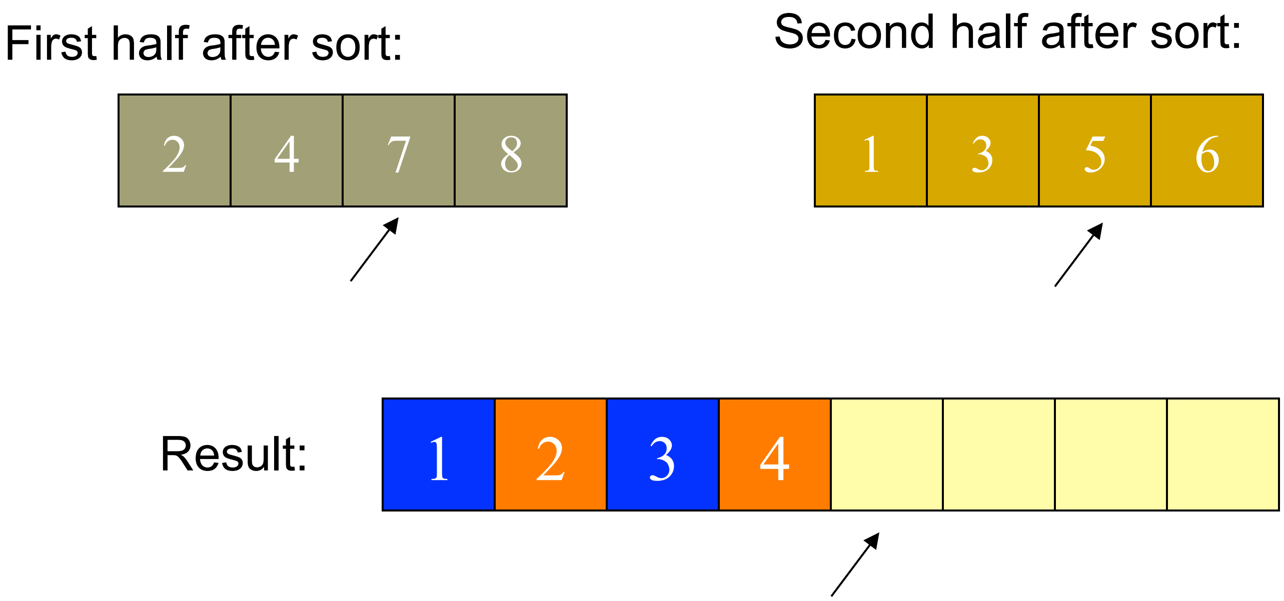
MERGE EXAMPLE

Merge operation: Use 3 pointers and 1 more array



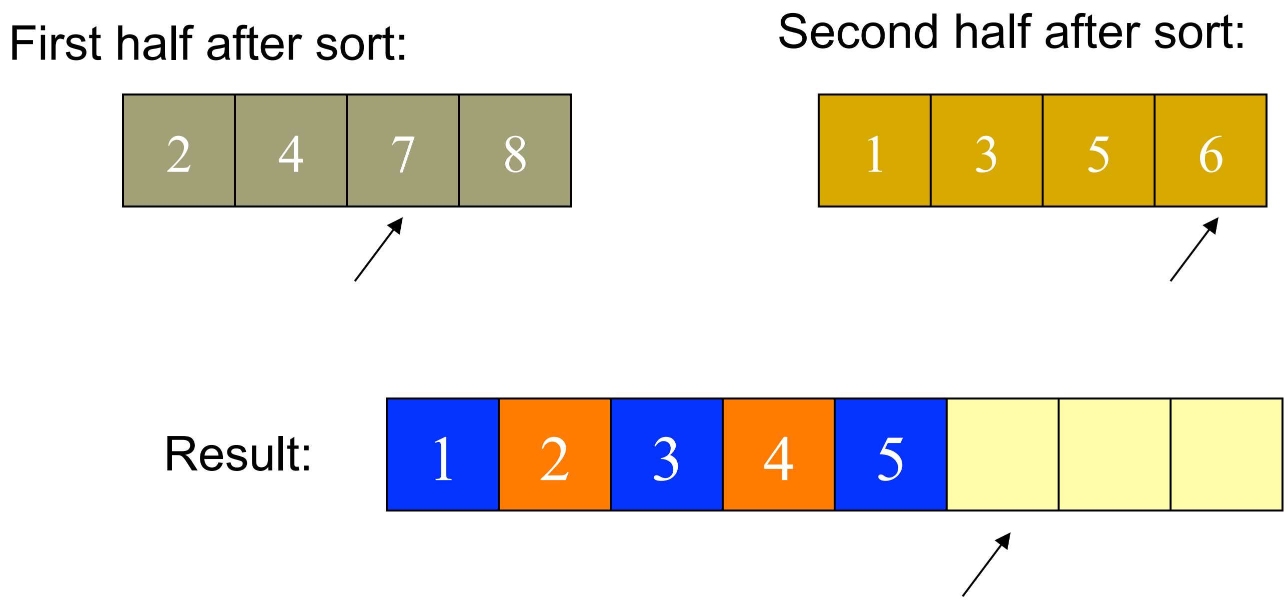
MERGE EXAMPLE

Merge operation: Use 3 pointers and 1 more array



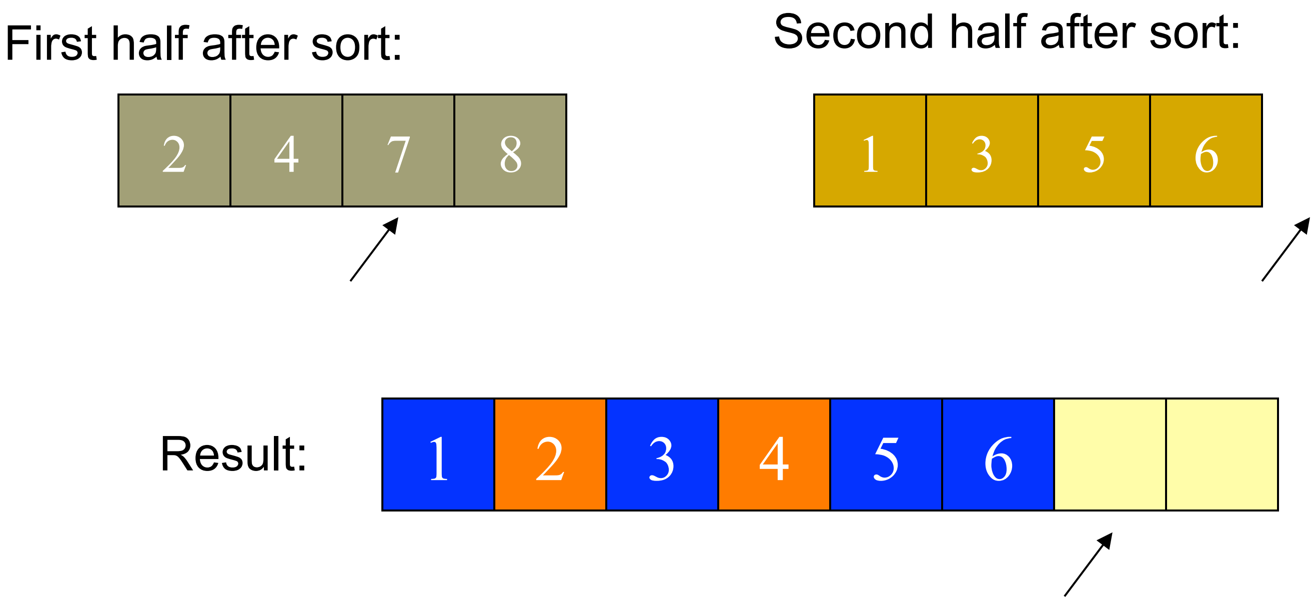
MERGE EXAMPLE

Merge operation: Use 3 pointers and 1 more array



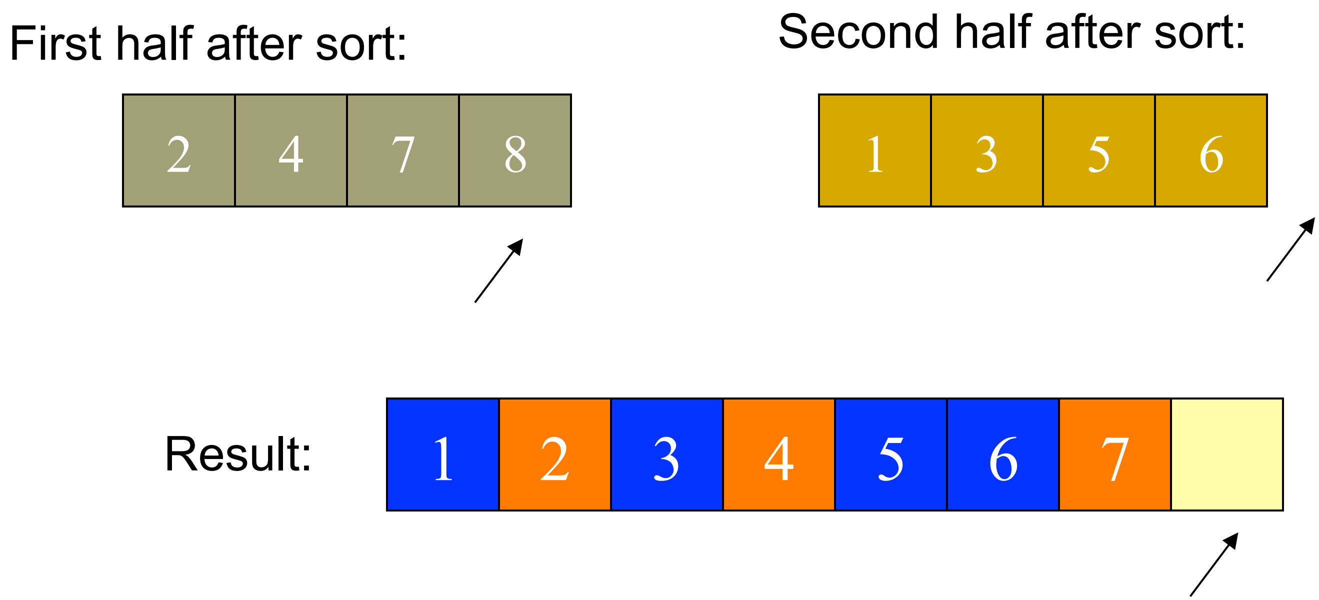
MERGE EXAMPLE

Merge operation: Use 3 pointers and 1 more array



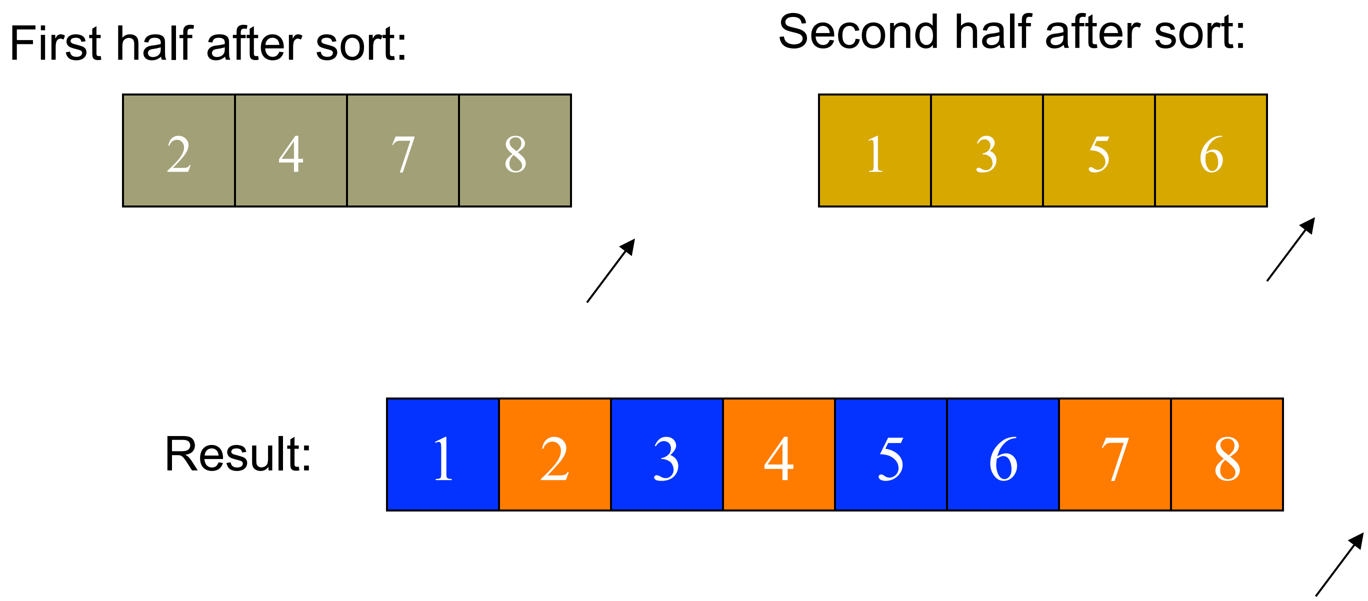
MERGE EXAMPLE

Merge operation: Use 3 pointers and 1 more array



MERGE EXAMPLE

Merge operation: Use 3 pointers and 1 more array

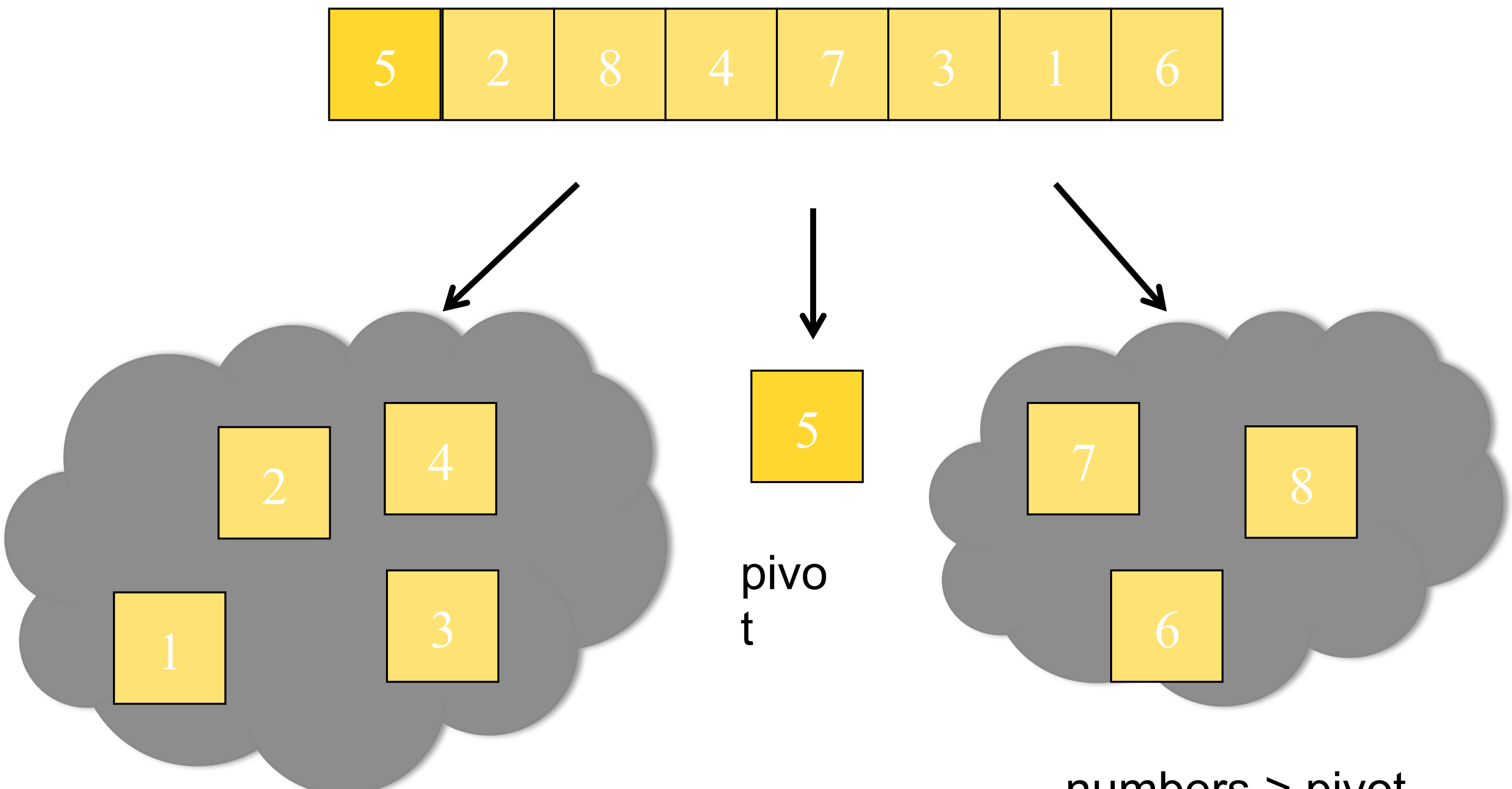


After Merge: copy result into original unsorted array.

Or alternate merging between two size n arrays.

QUICK SORT

Divide: Split array around a ‘pivot’



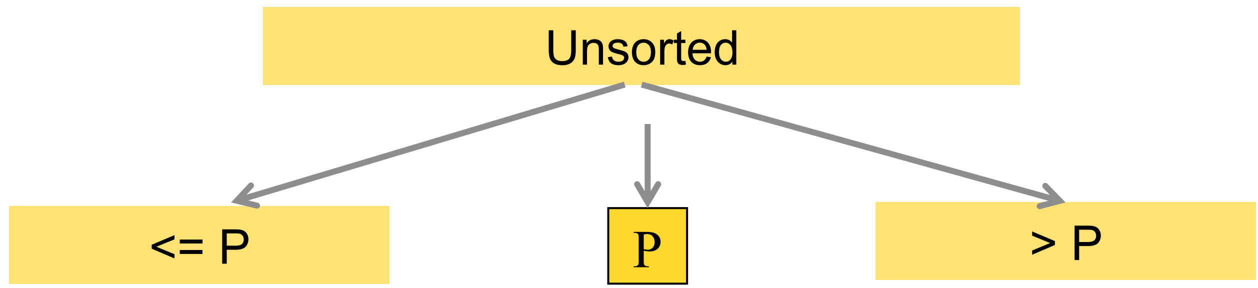
pivo
t

numbers > pivot

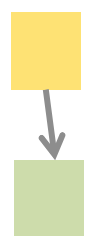
numbers <=
pivot

QUICK SORT

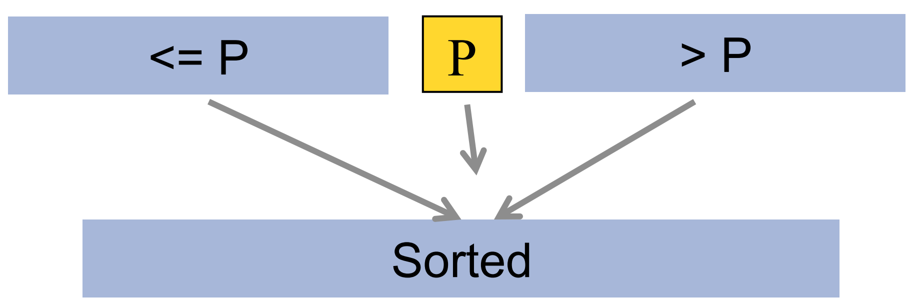
Divide: Pick a pivot, partition into groups



Conquer: Return array when length ≤ 1



Combine: Combine sorted partitions and pivot

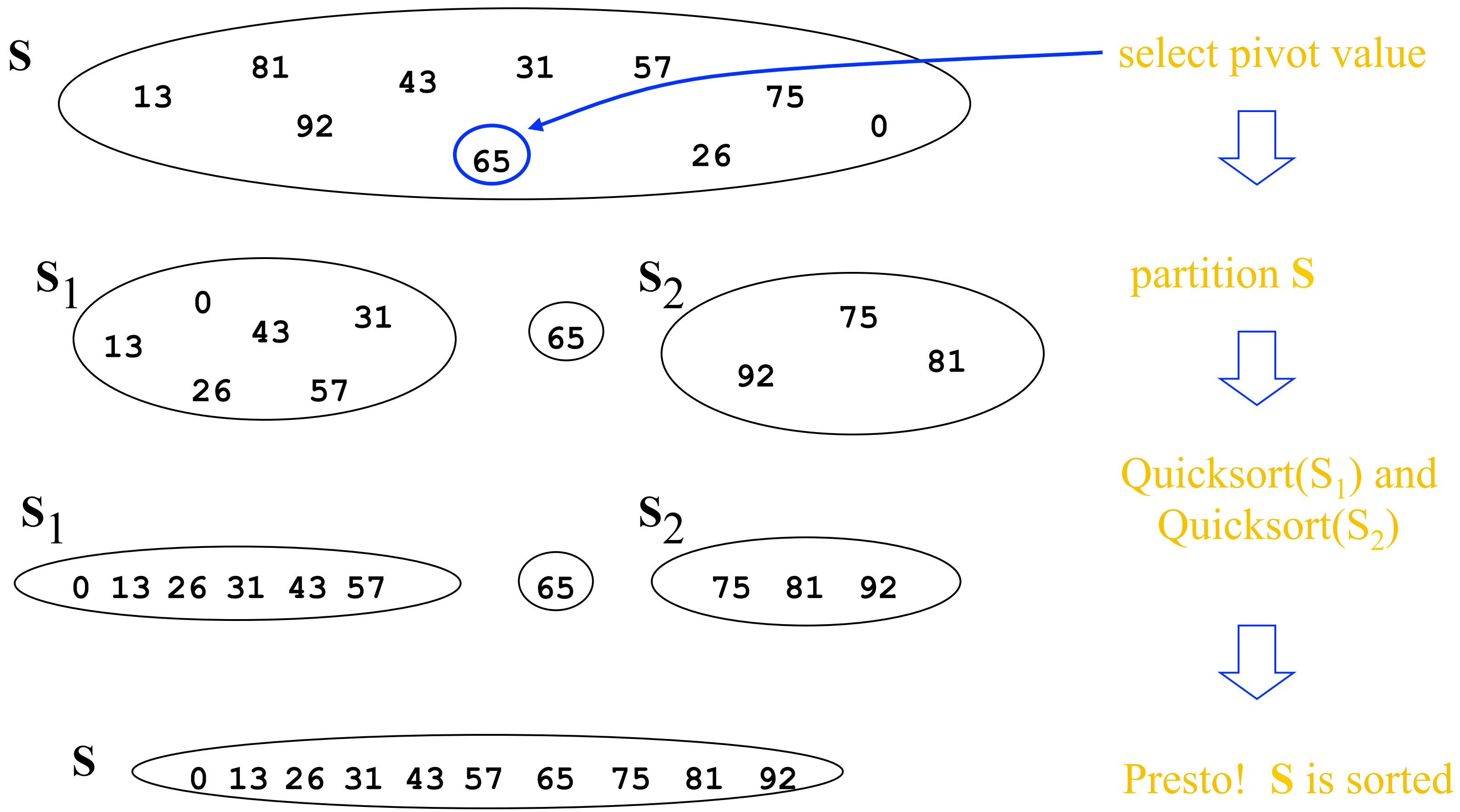


QUICK SORT PSEUDOCODE

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.

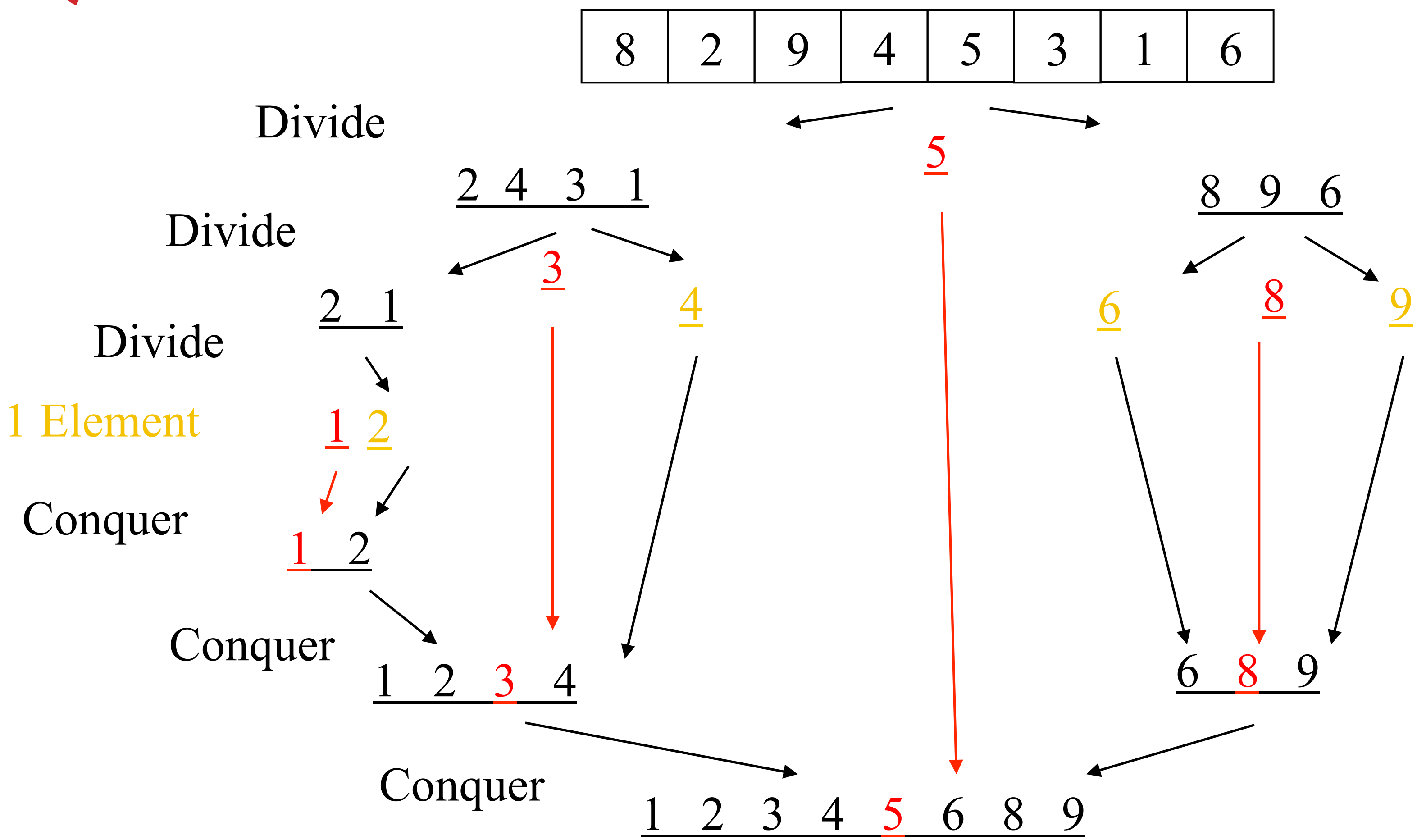
```
quicksort(input) {
    if (input.length < 2) {
        return input;
    } else {
        pivot = getPivot(input);
        smallerHalf = sort(getSmaller(pivot, input));
        largerHalf = sort(getBigger(pivot, input));
        return smallerHalf + pivot + largerHalf;
    }
}
```

QUICKSORT



[Weiss]

QUICKSORT

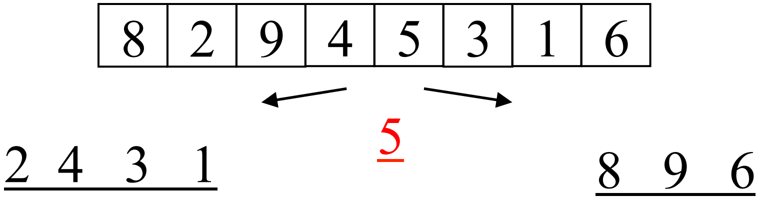


8 2 9 4 5 3 1 6
Divide
5
2 4 3 1
8 9 6
Divide
3
4
6
8
9
2 1
Divide
1 Element
1 2
Conquer
1 2
Conquer
1 2 3 4
6 8 9
Conquer
1 2 3 4 5 6 8 9

PIVOTS

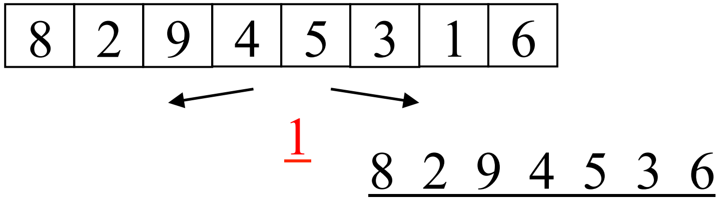
Best pivot?

- 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 counters **`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 cursors

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

Swap

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

Move cursors

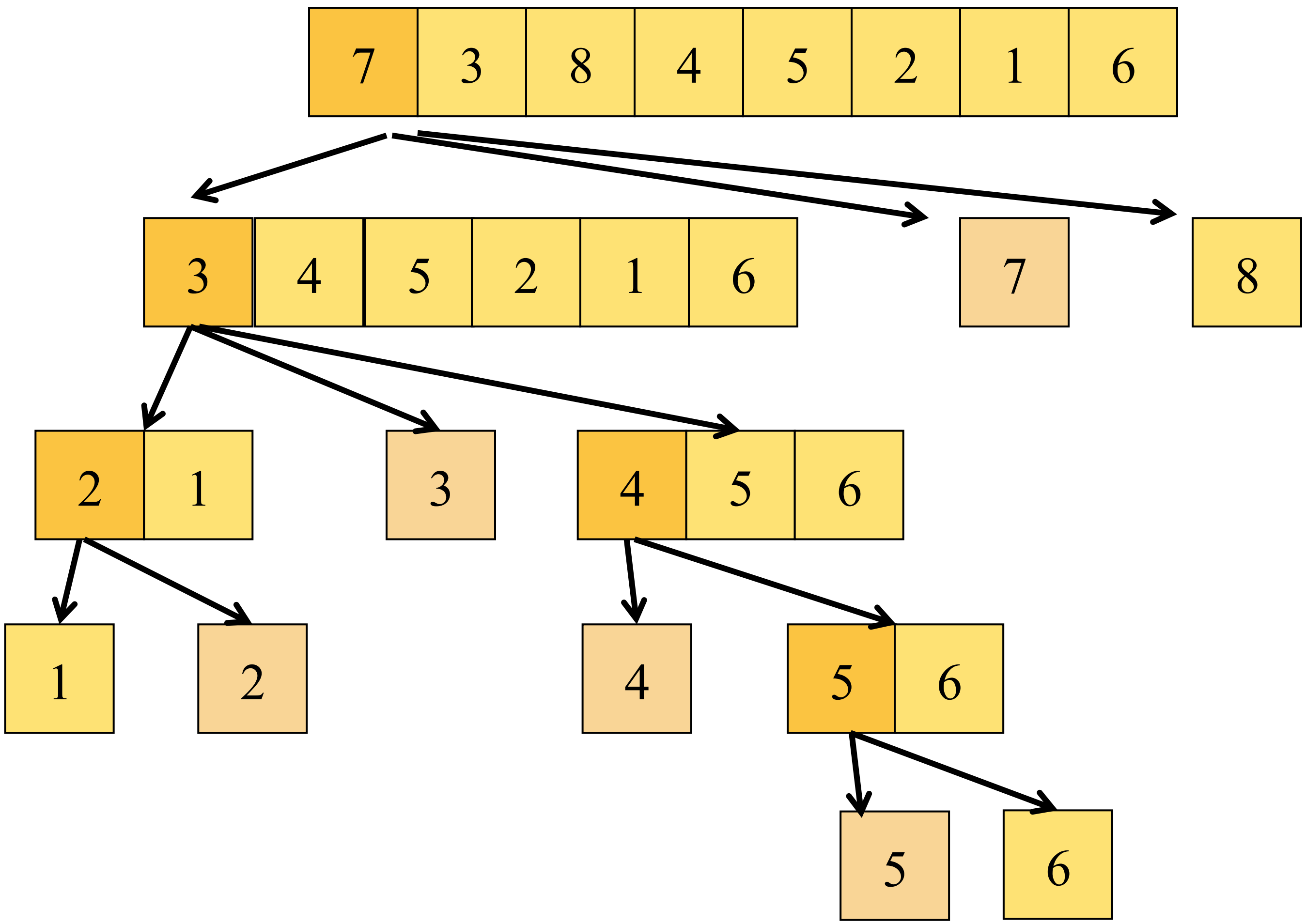
6	1	4	2	**0**	3	5	9	7	**8**

Move pivot

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

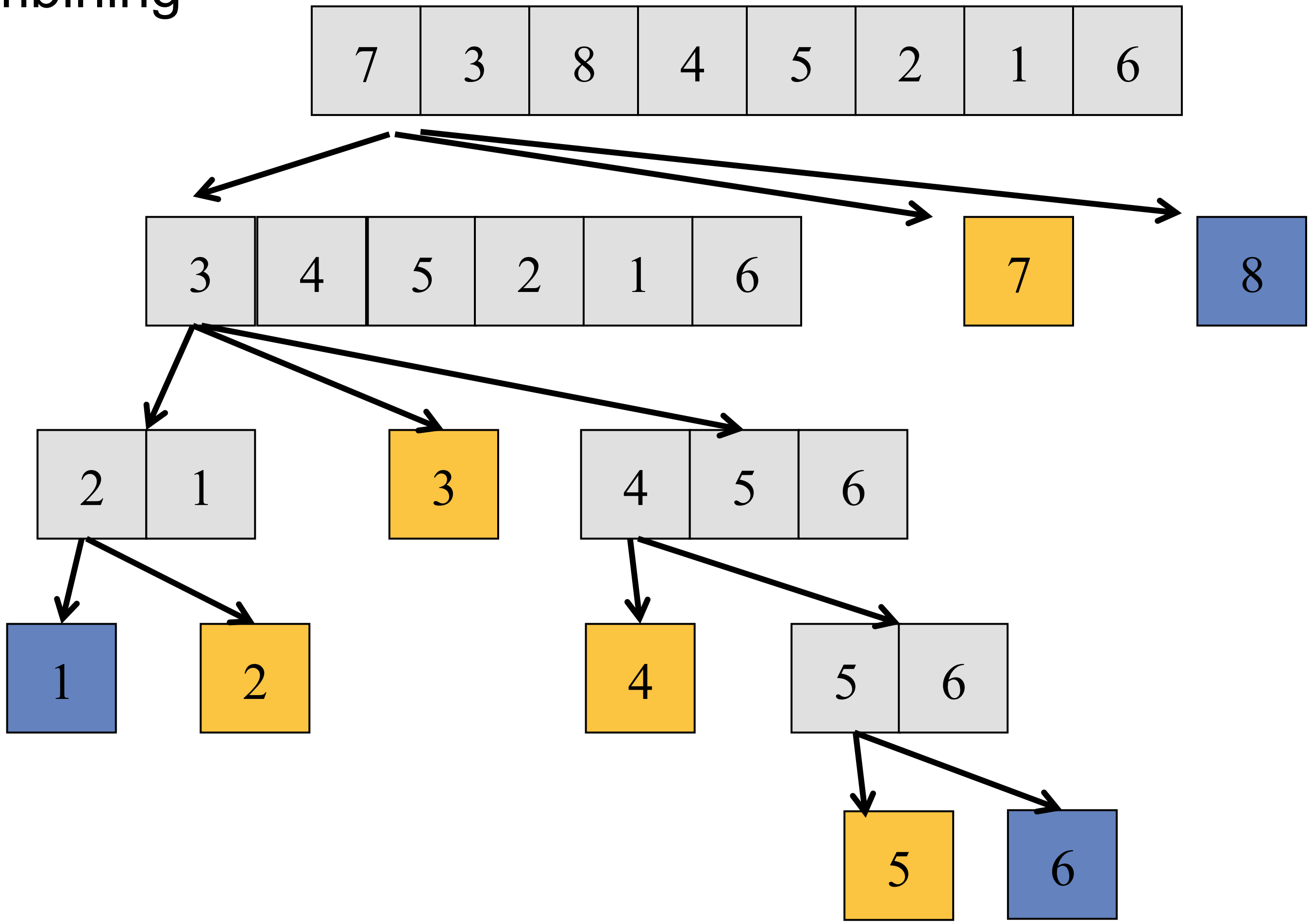
QUICK SORT EXAMPLE: DIVIDE

Pivot rule: pick the element at index 0



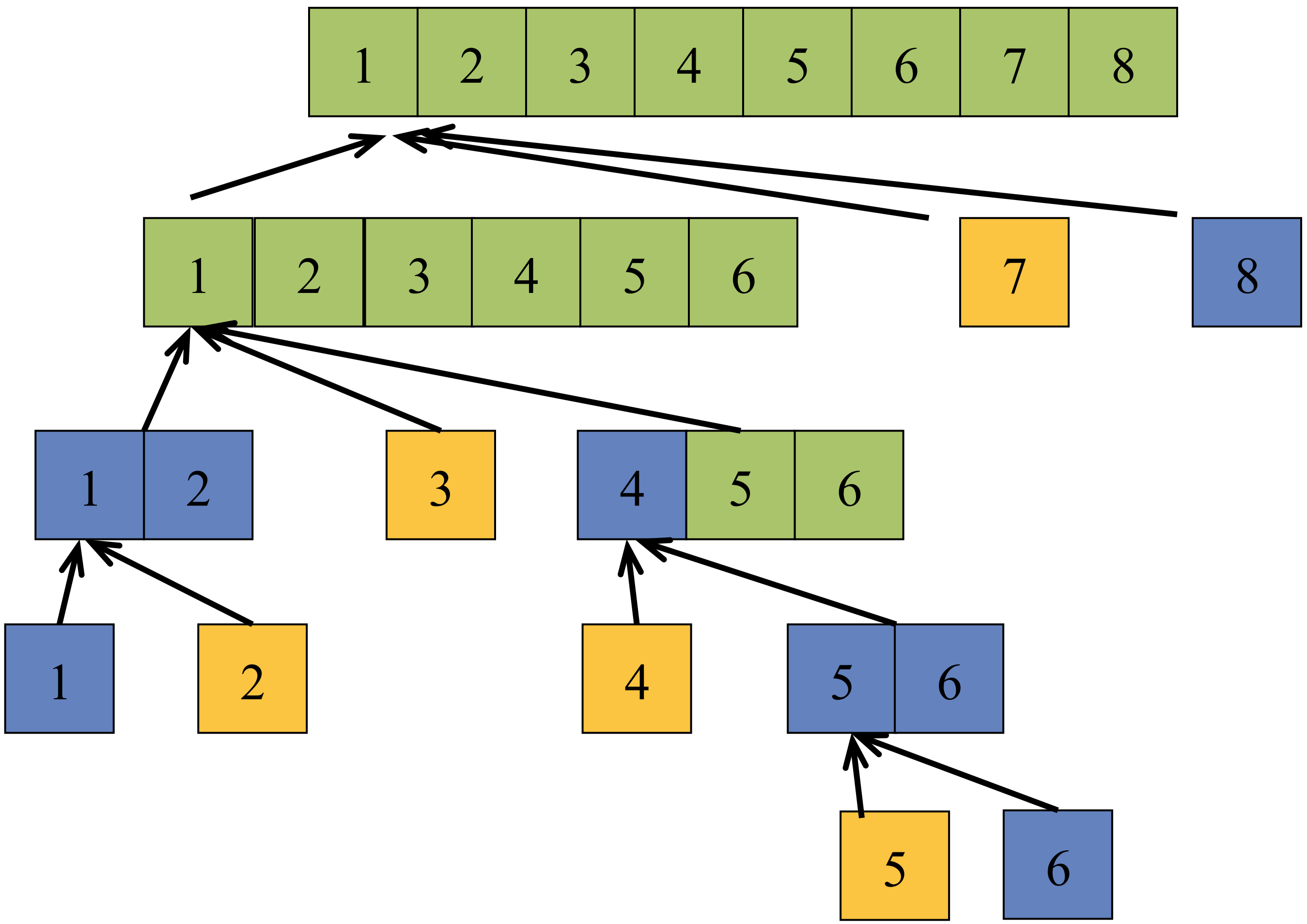
QUICK SORT EXAMPLE: COMBINE

Combine: this is the order of the elements we'll care about when combining



QUICK SORT EXAMPLE: COMBINE

Combine: put left partition < pivot < right partition



MEDIAN PIVOT EXAMPLE

Pick the median of first, middle, and last

7	2	8	4	5	3	1	6

Median = 6

Swap the median with the first value

7	2	8	4	5	3	1	6

Pivot is now at index 0, and we're ready to go

6	2	8	4	5	3	1	7

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. Put pivot in index `lo`
2. Use two pointers `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

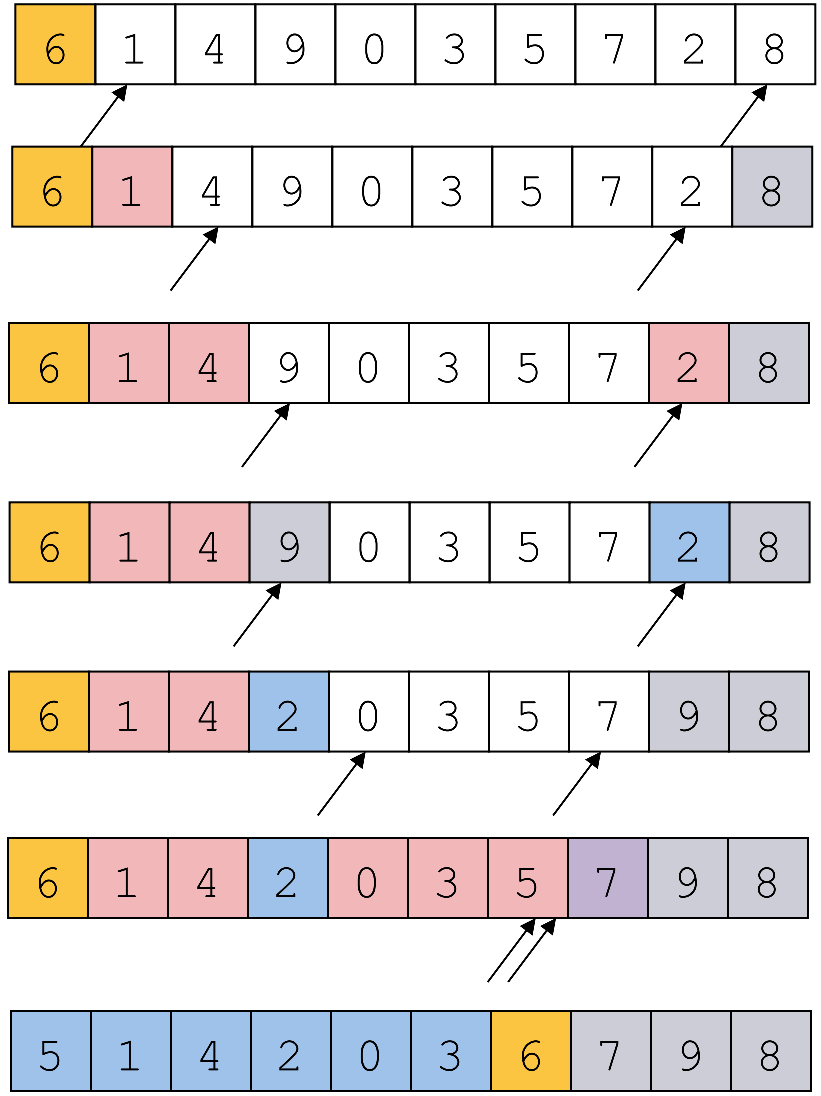
- **`lo`** = 0, **`hi`** = 10

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**

QUICK SORT PARTITION EXAMPLE



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

QUICK SORT 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)

- $O(n \log n)$, not responsible for proof

HOW FAST CAN WE SORT?

Heapsort & mergesort have $O(n \log n)$ worst-case running time

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

- ***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 $O(n \log \log n)$

COUNTING COMPARISONS

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

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

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*

COUNTING COMPARISONS

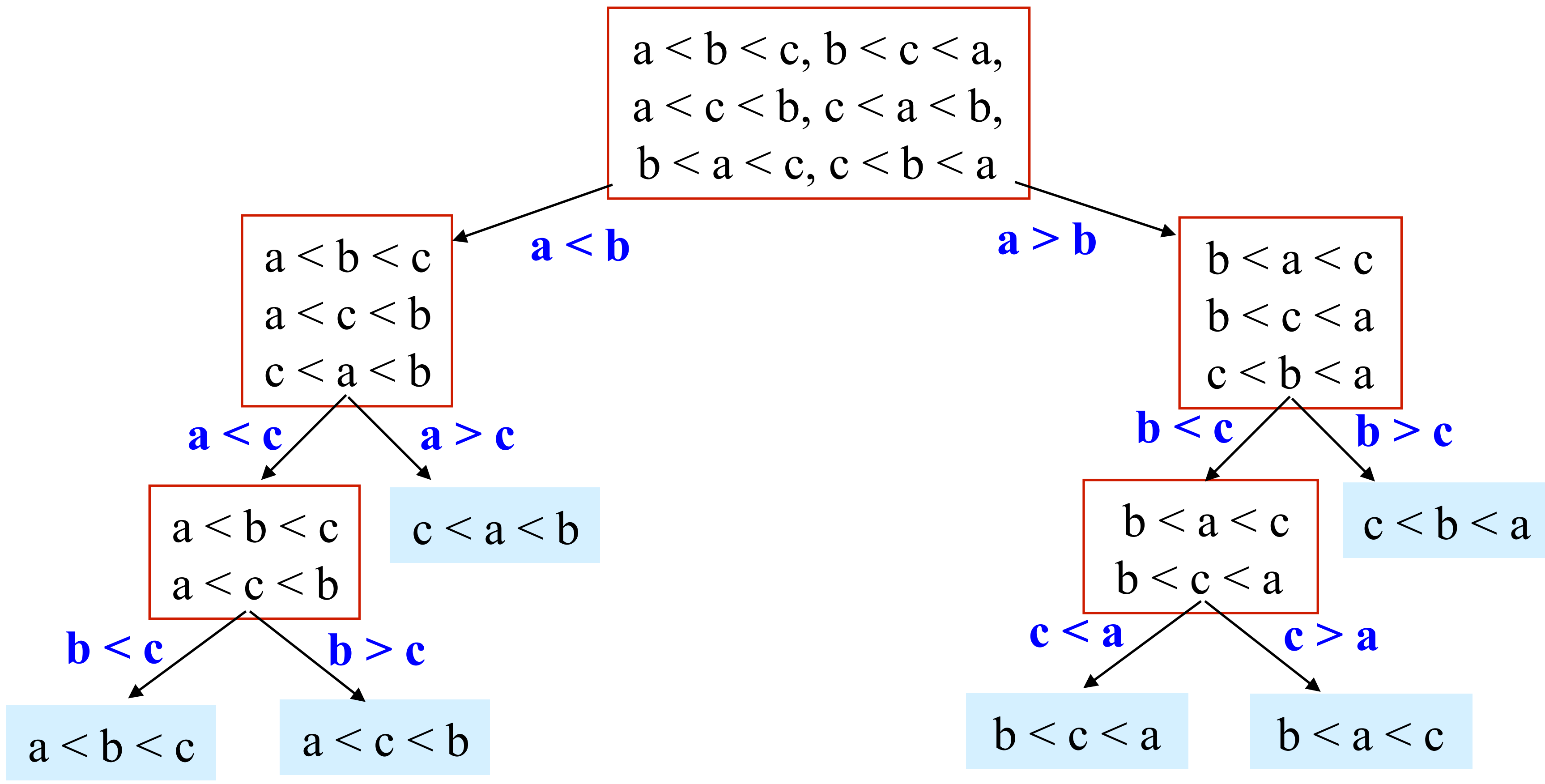
No matter what the algorithm is, it cannot make progress without doing comparisons

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

Can represent this process as a *decision tree*

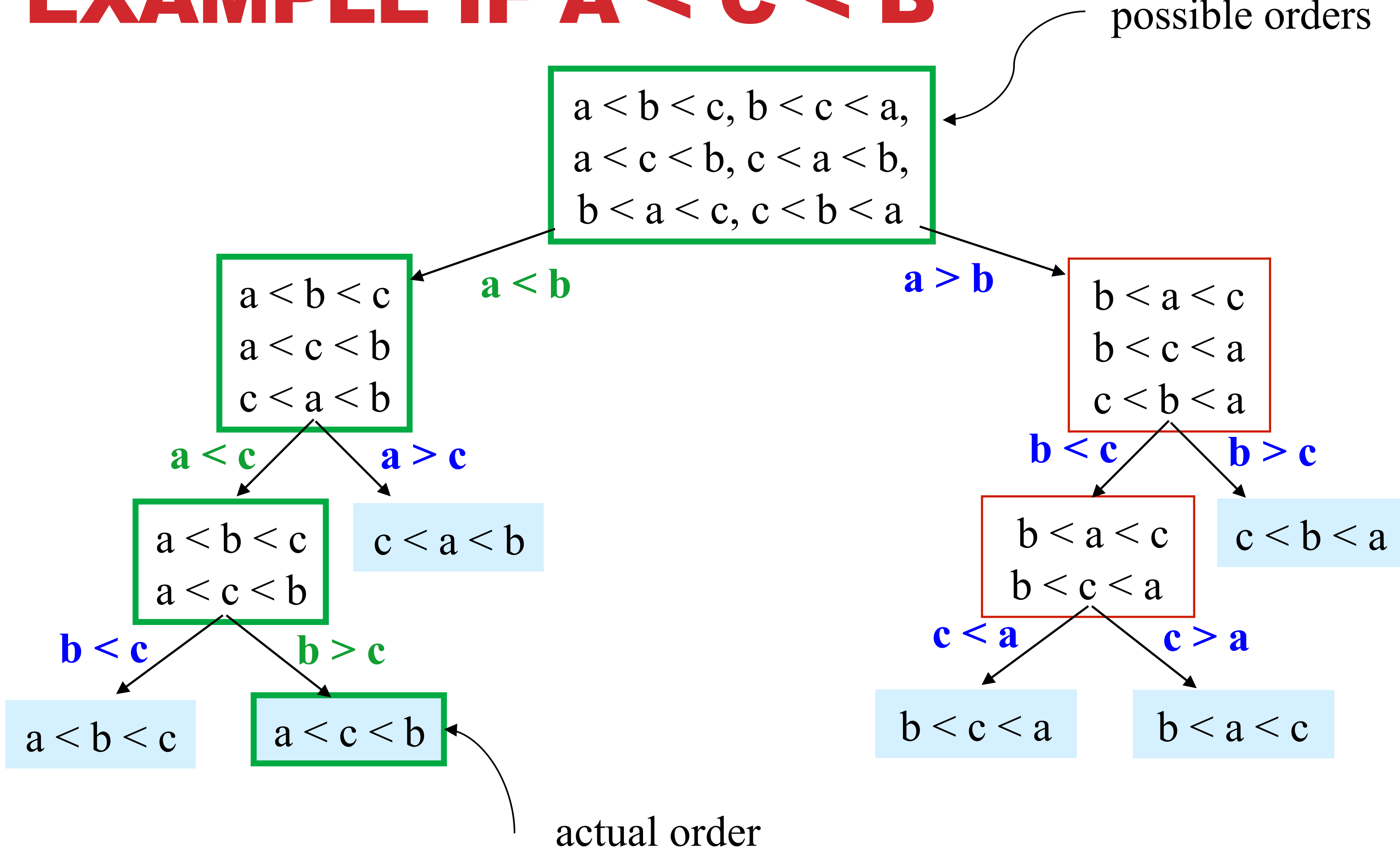
- Nodes contain "set of remaining possibilities"
- Edges are "answers from a comparison"
- The algorithm does not actually build the tree; it's what our *proof* uses to represent "the most the algorithm could know so far" as the algorithm progresses

DECISION TREE FOR N = 3



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

EXAMPLE IF A < C < B



DECISION TREE

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

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*-1)(*n*-2)…(2)(1) = ***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

RUNTIME

That proves runtime is at least $\Omega(\texttt{log}\ (n!))$. 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: $\Omega(n\,\texttt{log}\ n)$

SORTING

- **This is the lower bound for comparison sorts**

SORTING

- This is the lower bound for comparison sorts
- How can non-comparison sorts work better?

SORTING

- **This is the lower bound for comparison sorts**
- **How can non-comparison sorts work better?**
  - They need to know something about the data

SORTING

- **This is the lower bound for comparison sorts**
- **How can non-comparison sorts work better?**
  - They need to know something about the data
- **Strings and Ints are very well ordered**