CSE 373

NOVEMBER 8TH – COMPARISON SORTS

ASSORTED MINUTIAE

- **Bug in Project 3 files--reuploaded at midnight on Monday**
- **Project 2 scores**
  - Canvas groups is garbage – updated tonight
- **Extra credit**
  - P1 – done and feedback soon
  - P3 – EC posted to website
- **Midterm regrades – next Wednesday 12:00-2:00**

SORTING

- **Problem statement:**
  - Collection of Comparable data
  - Result should be a sorted collection of the data

SORTING

- **Problem statement:**
  - Collection of Comparable data
  - Result should be a sorted collection of the data
- **Motivation?**

SORTING

- **Problem statement:**
  - Collection of Comparable data
  - Result should be a sorted collection of the data
- **Motivation?**
  - Pre-processing v. find times
  - Sorting v. Maintaining sortedness

SORTING

- Important definitions

SORTING

- **Important definitions**
  - In-place: Requires only O(1) extra memory
    - **usually means the array is mutated**

SORTING

- **Important definitions**
  - In-place: Requires only O(1) extra memory
    - **usually means the array is mutated**
  - Stable: For any two elements have the same comparative value, then after the sort, which ever came first will stay first
    - Sorting by first name and then last name will give you **last then first** with a stable sort.
    - The most recent sort will always be the primary

SORTING

- **Important definitions**
    - Interruptable (top k): the algorithm can run only until the first *k* elements are in sorted order

SORTING

- **Important definitions**
  - Interruptable (top k): the algorithm can run only until the first *k* elements are in sorted order
  - Comparison sort: utilizes comparisons between elements to produce the final sorted order.

SORTING

- **Important definitions**
  - Interruptable (top k): the algorithm can run only until the first *k* elements are in sorted order
  - Comparison sort: utilizes comparisons between elements to produce the final sorted order.
    - Bogo sort is not a comparison sort

SORTING

- **Important definitions**
  - Interruptable (top k): the algorithm can run only until the first *k* elements are in sorted order
  - Comparison sort: utilizes comparisons between elements to produce the final sorted order.
    - Bogo sort is not a comparison sort
    - Comparison sorts are $\Omega(n \log n)$, they cannot do better than this

SORTING

- What are the sorts we’ve seen so far?

SORTING

- **What are the sorts we’ve seen so far?**
  - Selection sort:

SORTING

- **What are the sorts we’ve seen so far?**
  - Selection sort
    - Algorithm?

SORTING

- **What are the sorts we’ve seen so far?**
  - Selection sort
    - Algorithm? For each element, iterate through the array and select the lowest remaining element and place it at the end of the sorted portion.

SORTING

- **What are the sorts we’ve seen so far?**
  - Selection sort
    - Algorithm? For each element, iterate through the array and select the lowest remaining element and place it at the end of the sorted portion.
    - Runtime:

SORTING

- **What are the sorts we’ve seen so far?**
  - Selection sort
    - Algorithm? For each element, iterate through the array and select the lowest remaining element and place it at the end of the sorted portion.
    - Runtime:
      - First run, you must select from *n* elements, the second, from *n-1, and the kth* from *n-(k-1)*.

SORTING

- **What are the sorts we've seen so far?**
  - Selection sort
    - Algorithm? For each element, iterate through the array and select the lowest remaining element and place it at the end of the sorted portion.
    - Runtime:
      - First run, you must select from *n* elements, the second, from *n-1, and the kth* from *n-(k-1)*.
      - **What is this summation?** *n(n-1)/2*
    - Stable?

SORTING

- **What are the sorts we've seen so far?**
  - Selection sort
    - Algorithm? For each element, iterate through the array and select the lowest remaining element and place it at the end of the sorted portion.
    - Runtime:
      - First run, you must select from *n* elements, the second, from *n-1, and the kth* from *n-(k-1)*.
      - **What is this summation?** *n(n-1)/2*
    - Stable? **Not usually**

SORTING

- **What are the sorts we've seen so far?**
  - Selection sort
    - Algorithm? For each element, iterate through the array and select the lowest remaining element and place it at the end of the sorted portion.
    - Runtime:
      - First run, you must select from *n* elements, the second, from *n-1, and the kth* from *n-(k-1)*.
      - **What is this summation?** *n(n-1)/2*
    - Stable? How?

SORTING

- **What are the sorts we've seen so far?**
  - Selection sort
    - Algorithm? For each element, iterate through the array and select the lowest remaining element and place it at the end of the sorted portion.
    - Runtime:
      - First run, you must select from *n* elements, the second, from *n-1, and the kth* from *n-(k-1)*.
      - **What is this summation?** *n(n-1)/2*
    - Stable? How?
      - When you have your lowest candidate, shift other candidates over (similar to bubble sort)

SORTING

- **What are the sorts we’ve seen so far?**
  - Selection sort
    - Algorithm? For each element, iterate through the array and select the lowest remaining element and place it at the end of the sorted portion.
    - Runtime:
      - First run, you must select from *n* elements, the second, from *n-1, and the kth* from *n-(k-1)*.
      - **What is this summation?** *n(n-1)/2*
    - Stable? How?
      - When you have your lowest candidate, shift other candidates over (similar to bubble sort)
    - In place?

SORTING

- **What are the sorts we've seen so far?**
  - Selection sort
    - Algorithm? For each element, iterate through the array and select the lowest remaining element and place it at the end of the sorted portion.
    - Runtime:
      - First run, you must select from *n* elements, the second, from *n-1, and the kth* from *n-(k-1)*.
      - **What is this summation?** *n(n-1)/2*
    - Stable? How?
      - When you have your lowest candidate, shift other candidates over (similar to bubble sort)
    - In place? Can be, but can also create a separate collection (if we only want the top 5, for example)

SORTING

- **What are the sorts we’ve seen so far?**
  - Insertion Sort:
    - Algorithm?

SORTING

- **What are the sorts we’ve seen so far?**
  - Insertion Sort:
    - Algorithm? Maintain a sorted portion at the beginning of the array. For each new element, we swap it into the sorted portion until it reaches it’s correct location

SORTING

- **What are the sorts we’ve seen so far?**
  - Insertion Sort:
    - Algorithm? Maintain a sorted portion at the beginning of the array. For each new element, we swap it into the sorted portion until it reaches it’s correct location
    - Runtime?

SORTING

- **What are the sorts we've seen so far?**
  - Insertion Sort:
    - Algorithm? Maintain a sorted portion at the beginning of the array. For each new element, we swap it into the sorted portion until it reaches it's correct location
    - Runtime?
      - Worst-case: $O(n^2)$ – what case is this?

SORTING

- **What are the sorts we’ve seen so far?**
  - Insertion Sort:
    - Algorithm? Maintain a sorted portion at the beginning of the array. For each new element, we swap it into the sorted portion until it reaches it’s correct location
    - Runtime?
      - Worst-case: $O(n^2)$ – reverse sorted order

SORTING

- **What are the sorts we’ve seen so far?**
    - Insertion Sort:
        - Algorithm? Maintain a sorted portion at the beginning of the array. For each new element, we swap it into the sorted portion until it reaches it’s correct location
        - Runtime?
            - Worst-case: $O(n^2)$ – reverse sorted order
            - Best-case:

SORTING

- **What are the sorts we’ve seen so far?**
  - Insertion Sort:
    - Algorithm? Maintain a sorted portion at the beginning of the array. For each new element, we swap it into the sorted portion until it reaches it’s correct location
    - Runtime?
      - Worst-case: $O(n^2)$ – reverse sorted order
      - Best-case: $O(n)$

SORTING

- **What are the sorts we've seen so far?**
  - Insertion Sort:
    - Algorithm? Maintain a sorted portion at the beginning of the array. For each new element, we swap it into the sorted portion until it reaches it's correct location
    - Runtime?
      - Worst-case: O(n^2) – reverse sorted order
      - Best-case: O(n) – sorted order

SORTING

- **What are the sorts we've seen so far?**
  - Insertion Sort:
    - Algorithm? Maintain a sorted portion at the beginning of the array. For each new element, we swap it into the sorted portion until it reaches it's correct location
    - Runtime?
      - Worst-case: $O(n^2)$ – reverse sorted order
      - Best-case: $O(n)$ – sorted order
      - Where does this difference come from?

SORTING

- **What are the sorts we’ve seen so far?**
  - Insertion Sort:
    - Algorithm? Maintain a sorted portion at the beginning of the array. For each new element, we swap it into the sorted portion until it reaches it’s correct location
    - Runtime?
      - Worst-case: $O(n^2)$ – reverse sorted order
      - Best-case: $O(n)$ – sorted order
      - Where does this difference come from?
        - When “swapping” into the sorted array, it can stop when it reaches the correct position, possibly terminating early. Selection sort must check all *k* elements to be sure it has the correct one

SORTING

- **What are the sorts we've seen so far?**
  - Insertion Sort:
    - Algorithm? Maintain a sorted portion at the beginning of the array. For each new element, we swap it into the sorted portion until it reaches it's correct location
    - Runtime?
      - Worst-case: $O(n^2)$ – reverse sorted order
      - Best-case: $O(n)$ – sorted order
      - Where does this difference come from?
        - When "swapping" into the sorted array, it can stop when it reaches the correct position, possibly terminating early. Selection sort must check all *k* elements to be sure it has the correct one
    - Stable?

SORTING

- **What are the sorts we’ve seen so far?**
  - Insertion Sort:
    - Algorithm? Maintain a sorted portion at the beginning of the array. For each new element, we swap it into the sorted portion until it reaches it’s correct location
    - Runtime?
      - Worst-case: $O(n^2)$ – reverse sorted order
      - Best-case: $O(n)$ – sorted order
      - Where does this difference come from?
        - When “swapping” into the sorted array, it can stop when it reaches the correct position, possibly terminating early. Selection sort must check all *k* elements to be sure it has the correct one
    - Stable? Yes, if we maintain sorted order in case of ties.

SORTING

- **What are the sorts we’ve seen so far?**
  - Insertion Sort:
    - Algorithm? Maintain a sorted portion at the beginning of the array. For each new element, we swap it into the sorted portion until it reaches it’s correct location
    - Runtime?
      - Worst-case: $O(n^2)$ – reverse sorted order
      - Best-case: $O(n)$ – sorted order
      - Where does this difference come from?
        - When “swapping” into the sorted array, it can stop when it reaches the correct position, possibly terminating early. Selection sort must check all *k* elements to be sure it has the correct one
    - Stable? Yes, if we maintain sorted order in case of ties.
    - In-place?

SORTING

- **What are the sorts we've seen so far?**
  - Insertion Sort:
    - Algorithm? Maintain a sorted portion at the beginning of the array. For each new element, we swap it into the sorted portion until it reaches it's correct location
    - Runtime?
      - Worst-case: $O(n^2)$ – reverse sorted order
      - Best-case: $O(n)$ – sorted order
      - Where does this difference come from?
        - When "swapping" into the sorted array, it can stop when it reaches the correct position, possibly terminating early. Selection sort must check all *k* elements to be sure it has the correct one
    - Stable? Yes, if we maintain sorted order in case of ties.
    - In-place? Can be easily. Since not interruptable, having a duplicate array is only necessary if you don't want the original array to be mutated

SORTING

- What other sorting techniques can we consider?

SORTING

- **What other sorting techniques can we consider?**
  - We know O(n log n) is possible. How do we do it?

SORTING

- **What other sorting techniques can we consider?**
    - We know O(n log n) is possible. How do we do it?
    - Heap sort works on principles we already know.

SORTING

- **What other sorting techniques can we consider?**
  - We know O(n log n) is possible. How do we do it?
  - Heap sort works on principles we already know.
    - Building a heap from an array takes O(n) time

SORTING

- **What other sorting techniques can we consider?**
  - We know O(n log n) is possible. How do we do it?
  - Heap sort works on principles we already know.
    - Building a heap from an array takes O(n) time
    - Removing the smallest element from the array takes O(log n)

SORTING

- **What other sorting techniques can we consider?**
  - We know O(n log n) is possible. How do we do it?
  - Heap sort works on principles we already know.
    - Building a heap from an array takes O(n) time
    - Removing the smallest element from the array takes O(log n)
    - There are n elements.

SORTING

- **What other sorting techniques can we consider?**
  - We know O(n log n) is possible. How do we do it?
  - Heap sort works on principles we already know.
    - Building a heap from an array takes O(n) time
    - Removing the smallest element from the array takes O(log n)
    - There are n elements.
    - N + N*log N = O(N log N)

SORTING

- **What other sorting techniques can we consider?**
  - We know O(n log n) is possible. How do we do it?
  - Heap sort works on principles we already know.
    - Building a heap from an array takes O(n) time
    - Removing the smallest element from the array takes O(log n)
    - There are n elements.
    - N + N*log N = O(N log N)
    - Using Floyd's method does not improve the asymptotic runtime for heap sort, but it is an improvement.

HEAP SORT

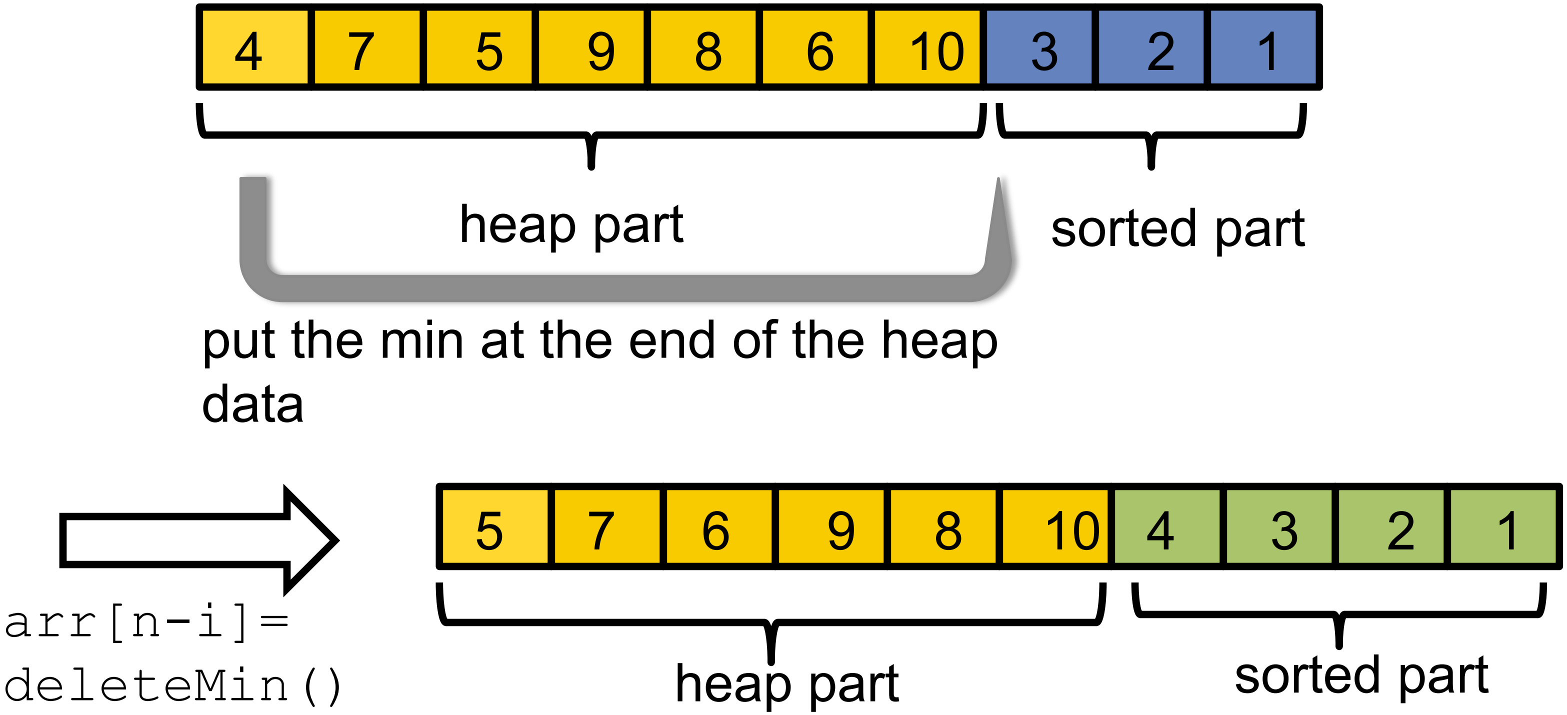
- How do we actually implement this sort?
- Can we do it in place?

HEAP SORT

- How do we actually implement this sort?
- Can we do it in place?

IN-PLACE HEAP SORT

- 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!



put the min at the end of the heap data

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

HEAP SORT

- How do we actually implement this sort?
- Can we do it in place?
- Is this sort stable?

HEAP SORT

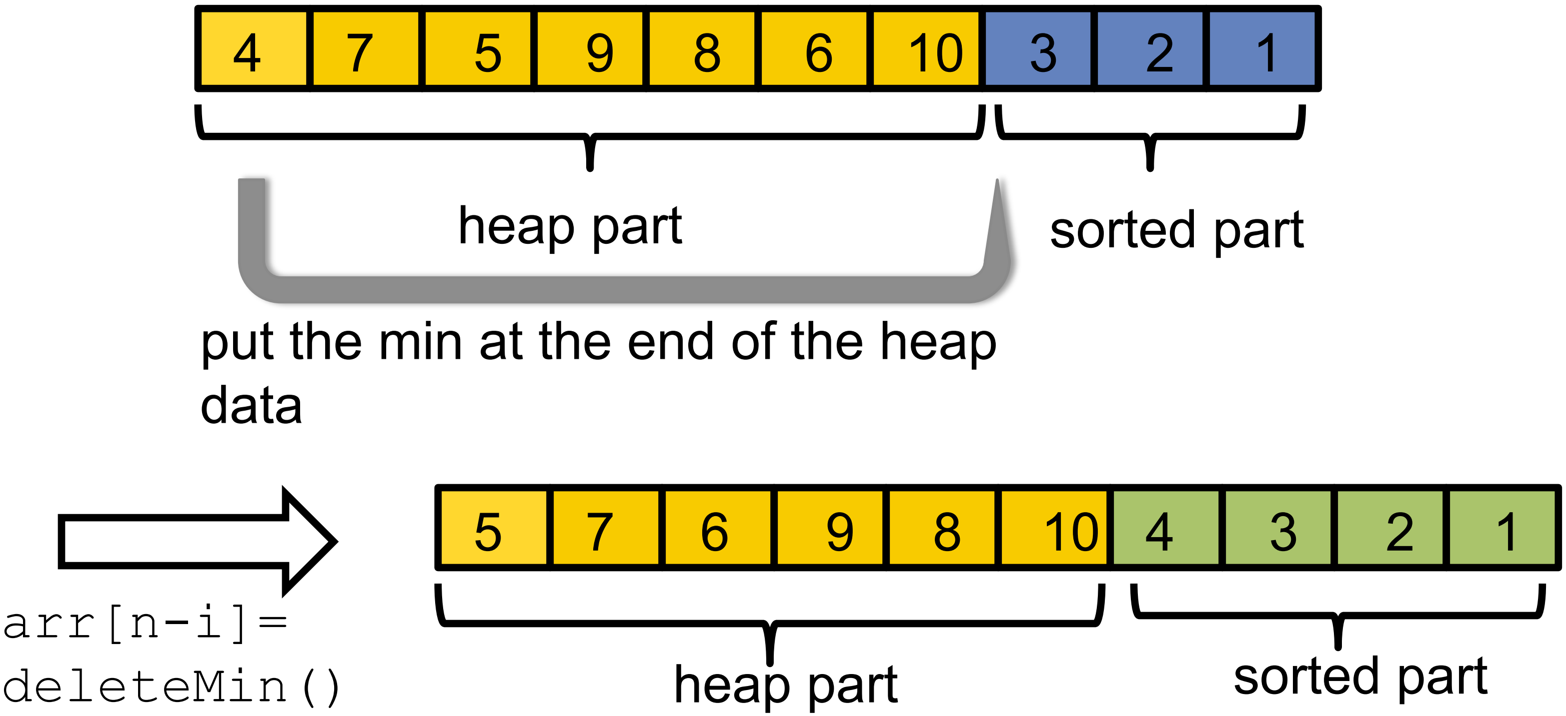
- **How do we actually implement this sort?**
- **Can we do it in place?**
- **Is this sort stable?**
  - No. Recall that heaps do not preserve FIFO property

HEAP SORT

- **How do we actually implement this sort?**
- **Can we do it in place?**
- **Is this sort stable?**
  - No. Recall that heaps do not preserve FIFO property
  - If it needed to be stable, we would have to modify the priority to indicate its place in the array, so that each element has a unique priority.

IN-PLACE HEAP SORT

What is undesirable about this method?



4	7	5	9	8	6	10	3	2	1

heap part

sorted part

put the min at the end of the heap data

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

5	7	6	9	8	10	4	3	2	1

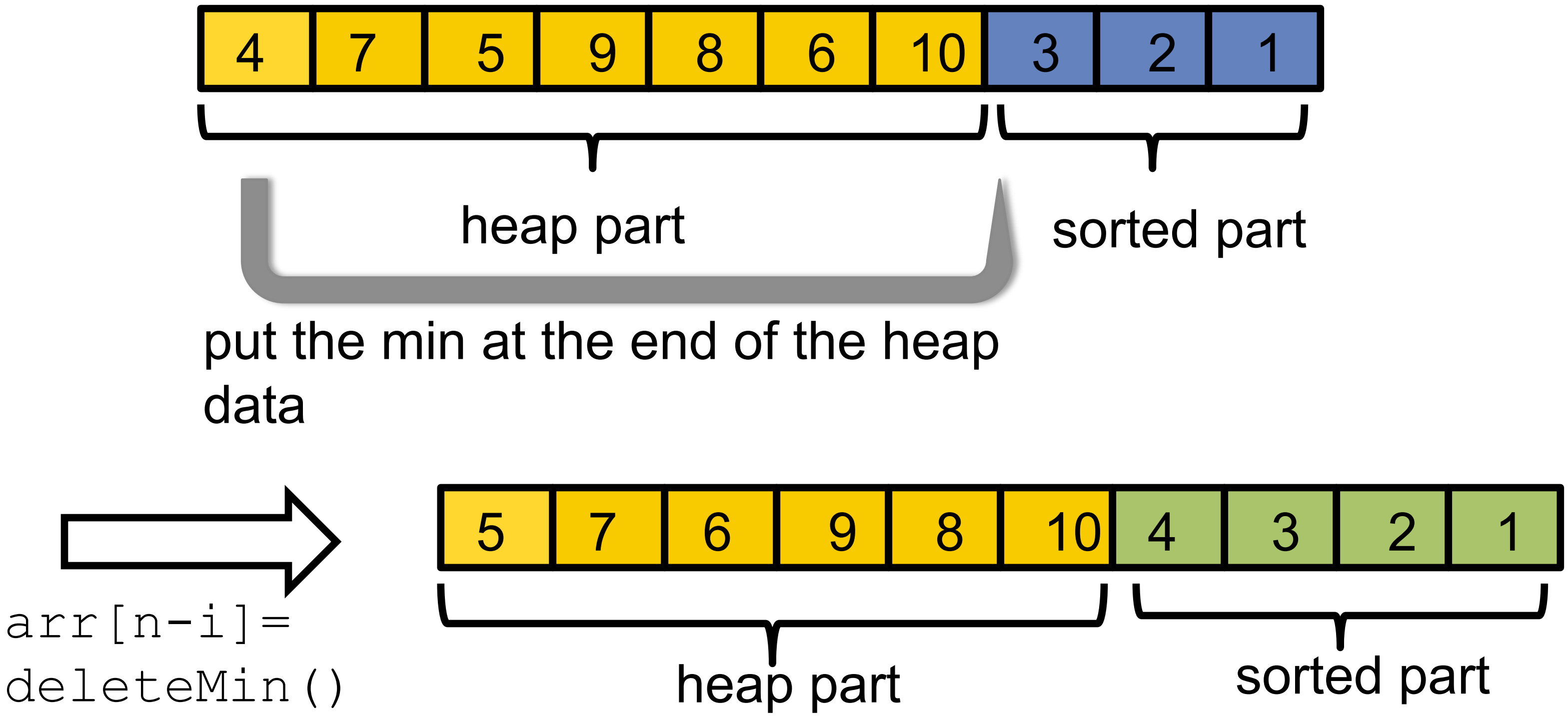
heap part

sorted part

IN-PLACE HEAP SORT

What is undesirable about this method?

You must reverse the array at the end.



4	7	5	9	8	6	10	3	2	1

heap part

sorted part

put the min at the end of the heap data

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

5	7	6	9	8	10	4	3	2	1

heap part

sorted part

HEAP SORT

- Can implement with a max-heap, then the sorted portion of the array fills in from the back and doesn’t need to be reversed at the end.

“AVL SORT”? “HASH SORT”?

AVL Tree: sure, we can also use an AVL tree to:

“AVL SORT”? “HASH SORT”?

AVL Tree: sure, we can also use an AVL tree to:

- **`insert`** each element: total time $O(n \texttt{ log } n)$
- Repeatedly **`deleteMin`**: total time $O(n \texttt{ log } n)$
    - Better: in-order traversal $O(n)$, but still $O(n \texttt{ log } n)$ overall
- But this cannot be done in-place and has worse constant factors than heap sort

“AVL SORT”? “HASH SORT”?

AVL Tree: sure, we can also use an AVL tree to:

- `insert` each element: total time $O(n \log n)$
- Repeatedly `deleteMin`: total time $O(n \log n)$
    - Better: in-order traversal $O(n)$, but still $O(n \log n)$ overall
- But this cannot be done in-place and has worse constant factors than heap sort

Hash Structure: don’t even think about trying to sort with a hash table!

“AVL SORT”? “HASH SORT”?

AVL Tree: sure, we can also use an AVL tree to:

- `insert` each element: total time $O(n \log n)$
- Repeatedly `deleteMin`: total time $O(n \log n)$
  - Better: in-order traversal $O(n)$, but still $O(n \log n)$ overall
- But this cannot be done in-place and has worse constant factors than heap sort

Hash Structure: don’t even think about trying to sort with a hash table!

- Finding min item in a hashtable is $O(n)$, so this would be a slower, more complicated selection 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

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:

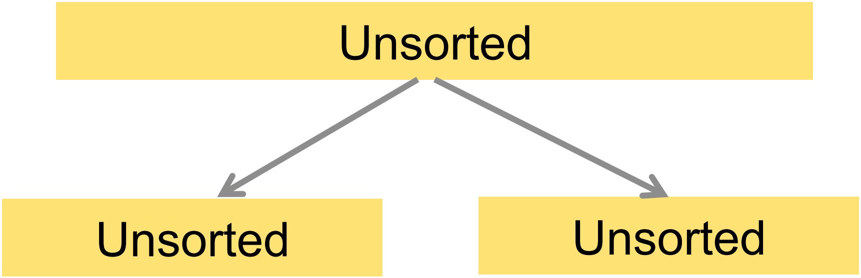
Sort the left half of the elements (recursively)
Sort the right half of the elements (recursively)
Merge the two sorted halves into a sorted whole

Quicksort:

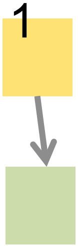
Pick a “pivot” element
Divide elements into less-than pivot and greater-than pivot
Sort the two divisions (recursively on each)
Answer is: sorted-less-than....pivot....sorted-greater-than

MERGE SORT

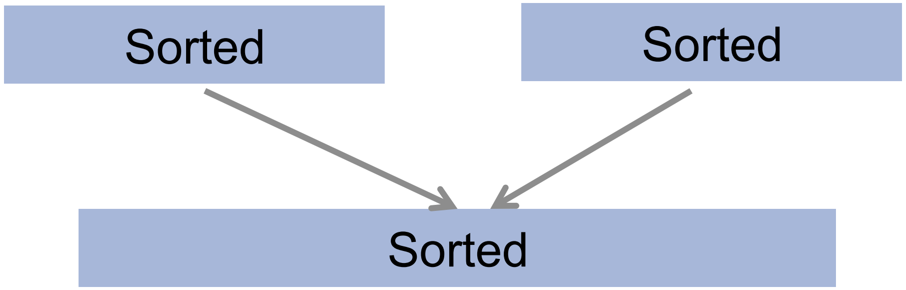
Divide: Split array roughly into half



Conquer: Return array when length ≤ 1



Combine: Combine two sorted arrays using merge

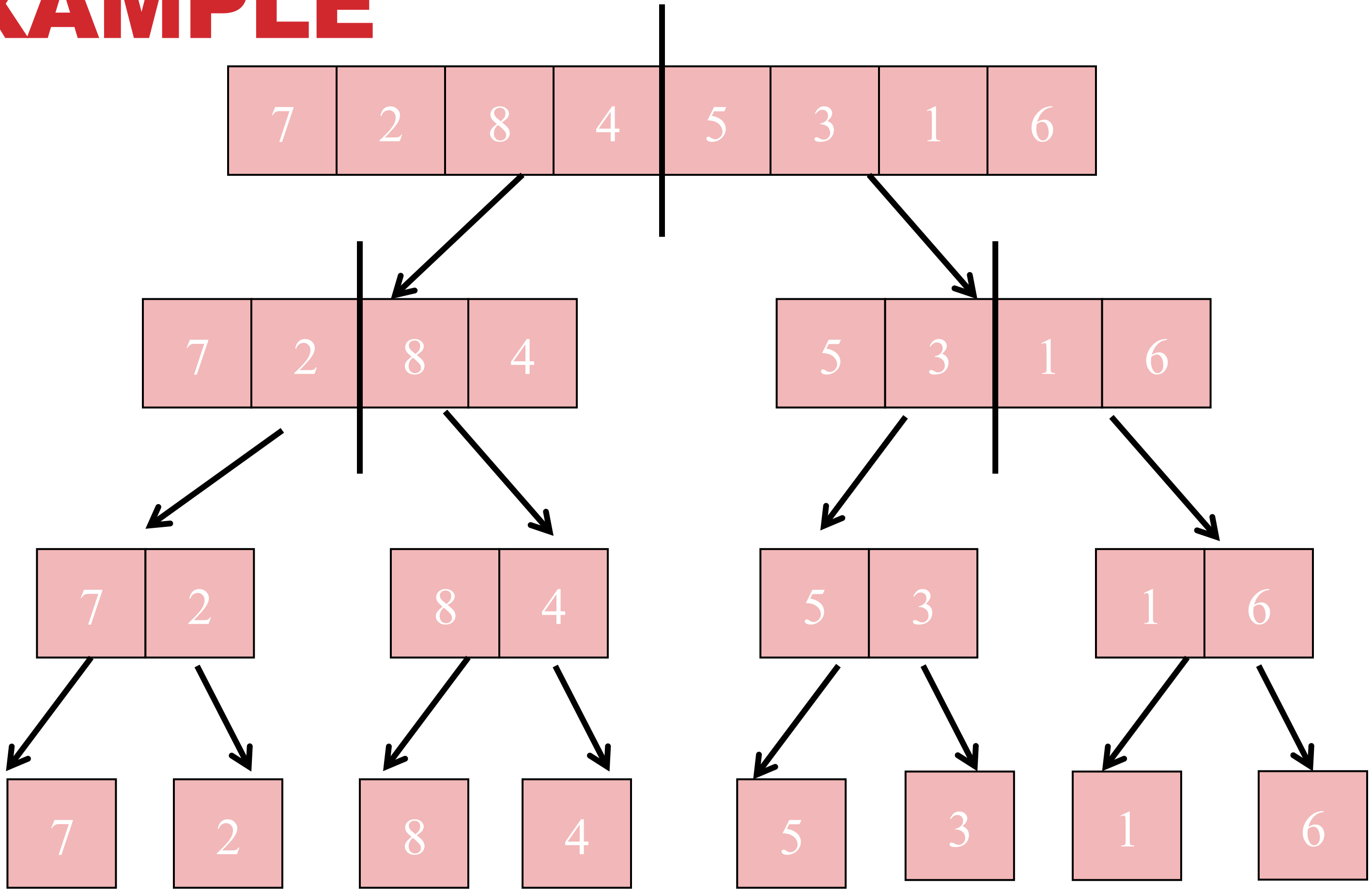


MERGE SORT: PSEUDOCODE

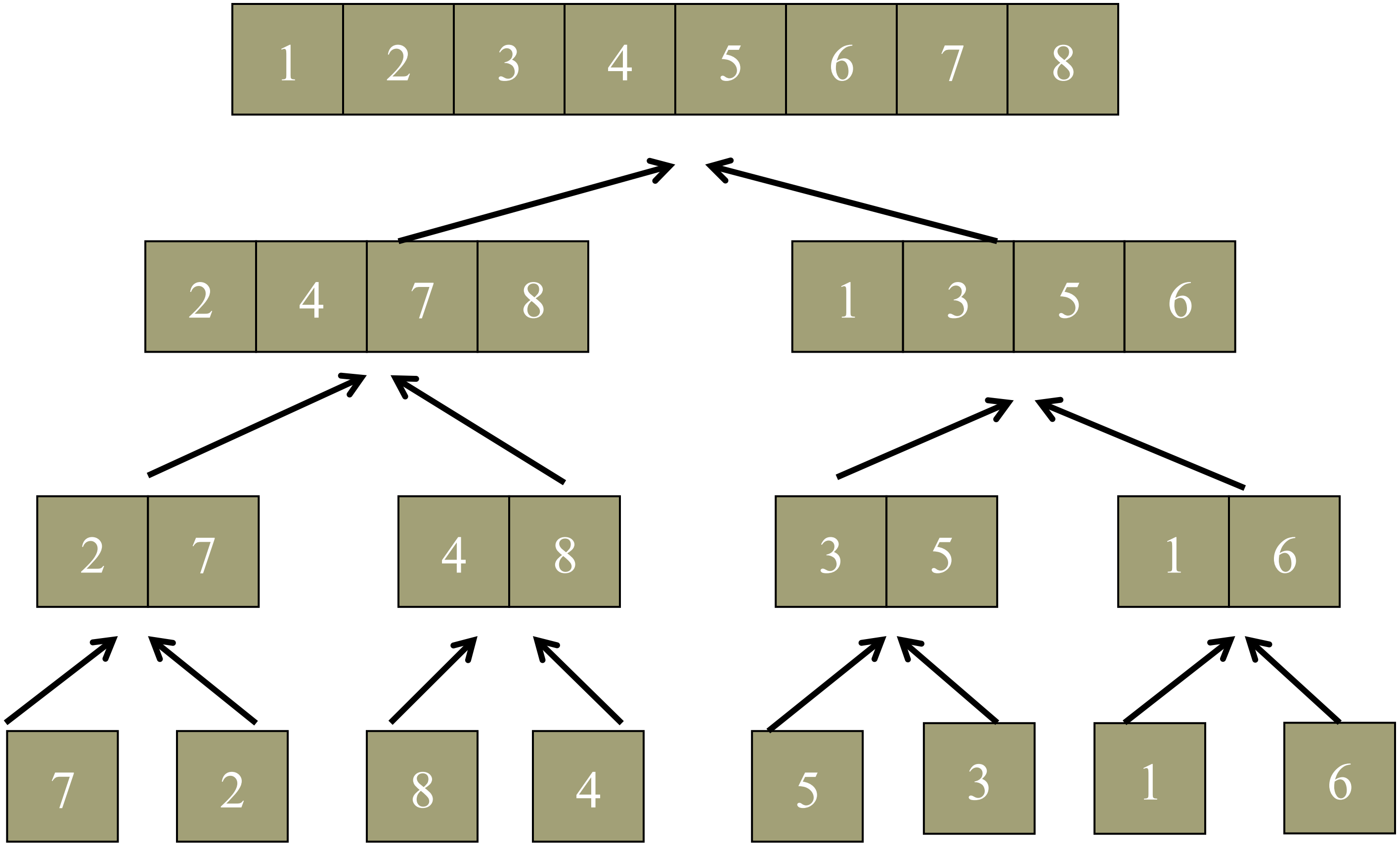
Core idea: split array in half, sort each half, merge back together. If the array has size 0 or 1, just return it unchanged

```
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

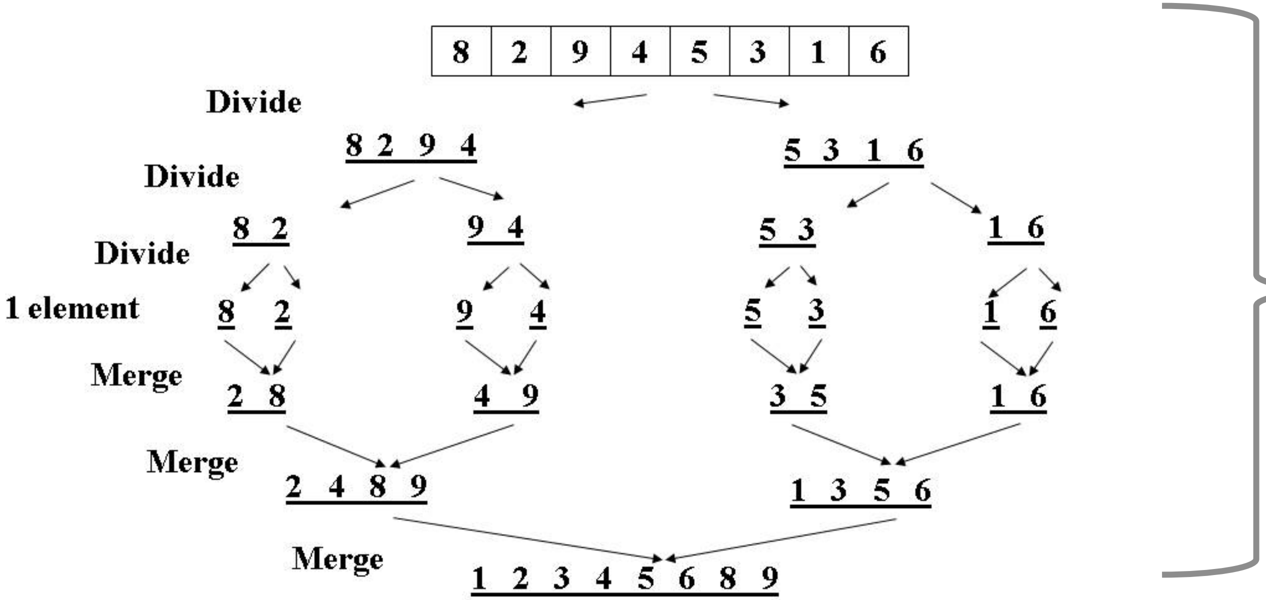


MERGE SORT ANALYSIS

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?

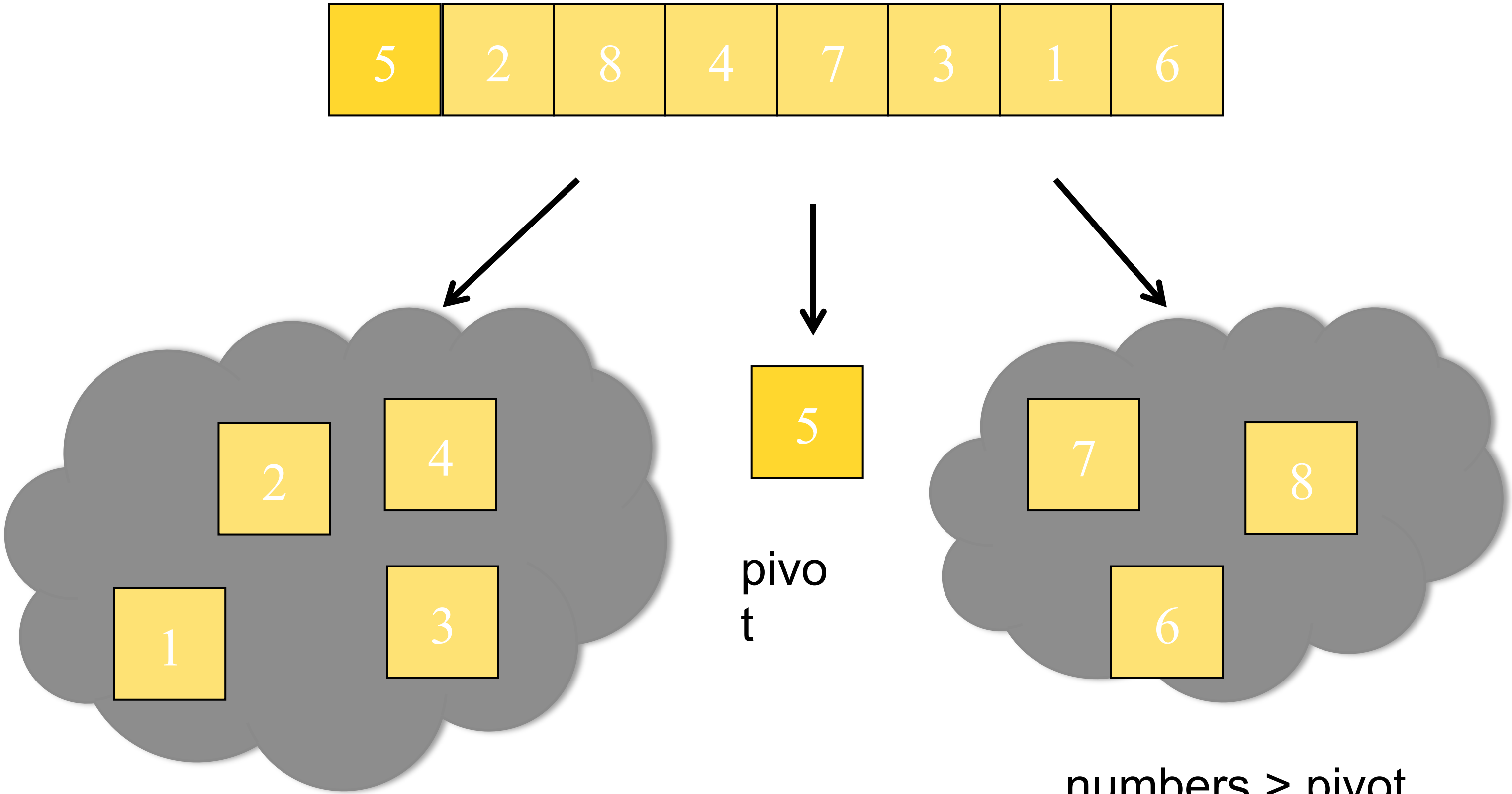
No. Merge must construct a new array to contain the output, so merge sort is not in-place.

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

One Solution: create a single auxiliary array and swap between it and the original on each level.

QUICK SORT

Divide: Split array around a ‘pivot’



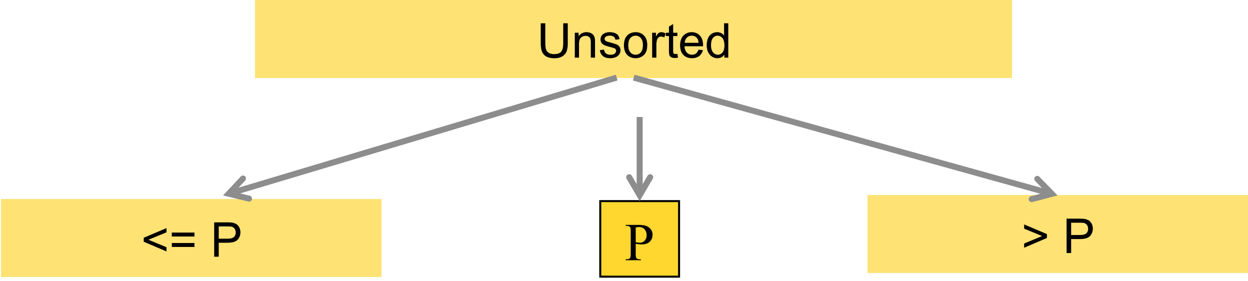
pivot

numbers <= pivot

numbers > pivot

QUICK SORT

Divide: Pick a pivot, partition into groups



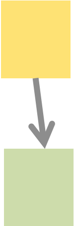
Unsorted

<= P

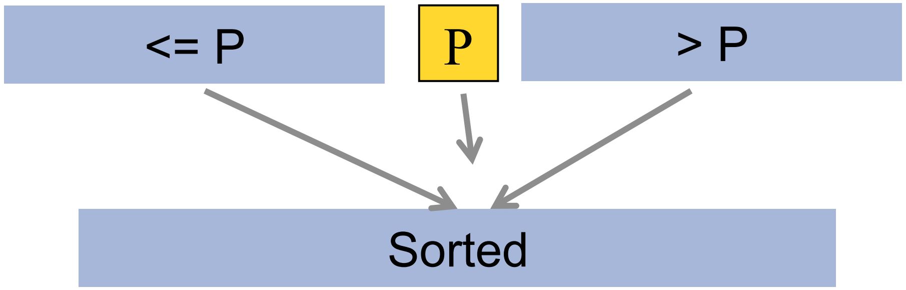
P

> P

Conquer: Return array when length ≤ 1



Combine: Combine sorted partitions and pivot



<= P

P

> P

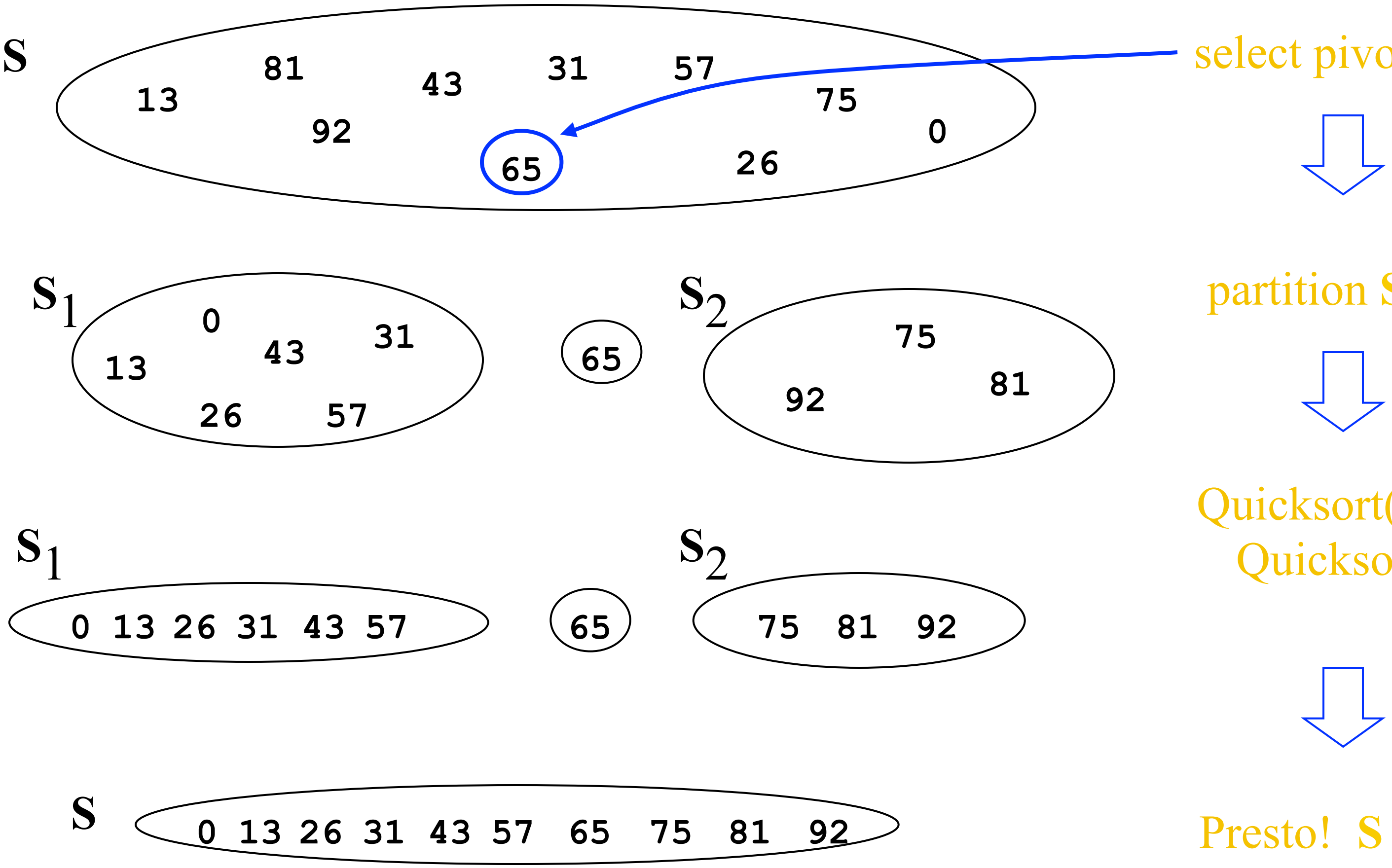
Sorted

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



select pivot value

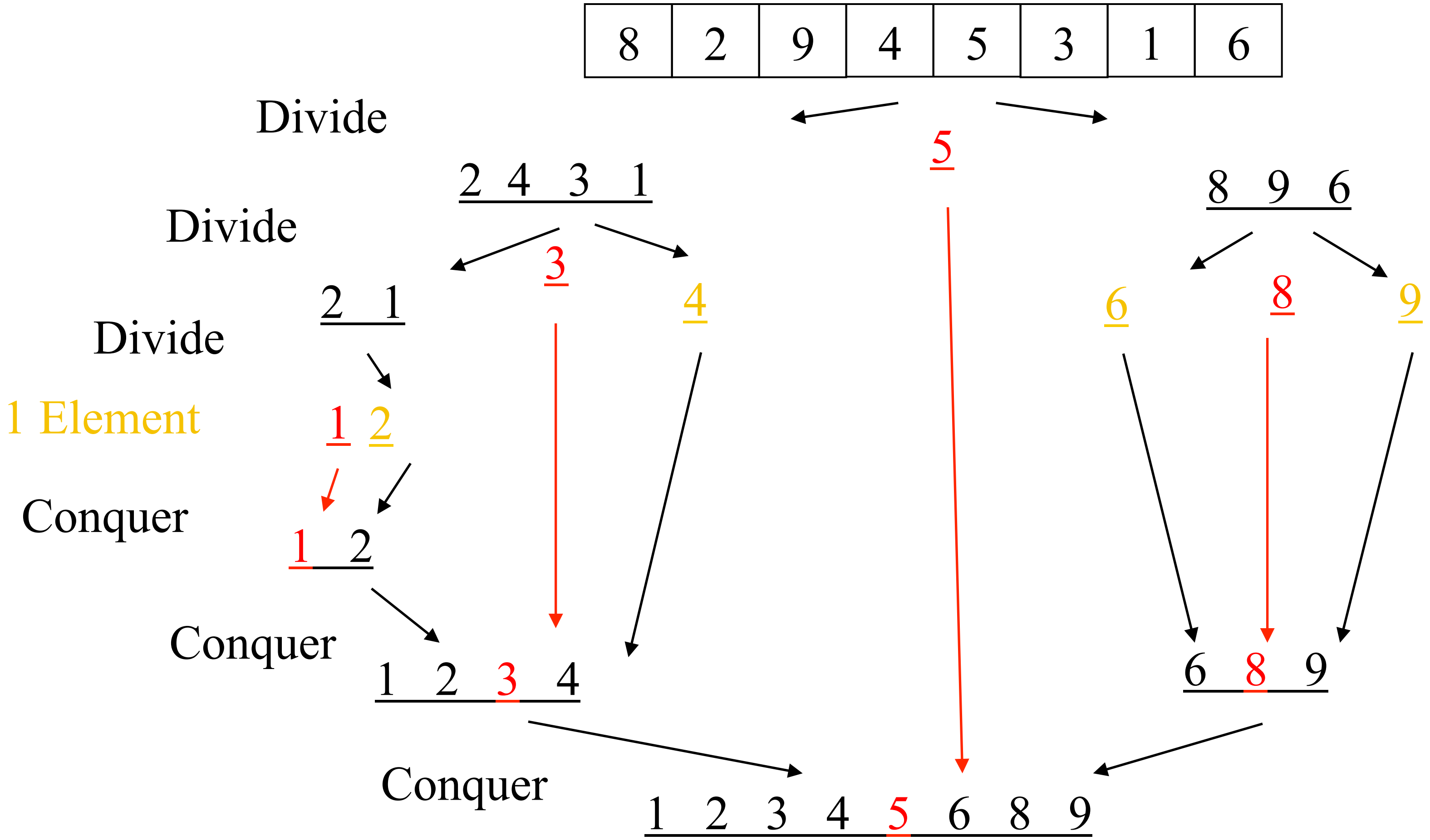
partition **S**

Quicksort(S_1) and
Quicksort(S_2)

Presto! **S** is sorted

[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

DETAILS

Have not yet explained:

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

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