CSE 332

JULY 19TH – SORTING 2

ASSORTED MINUTIAE

- Exams graded

ASSORTED MINUTIAE

- **Exams graded**
  - Should have in hand

ASSORTED MINUTIAE

- **Exams graded**
  - Should have in hand
- **Projects graded**

ASSORTED MINUTIAE

- **Exams graded**
  - Should have in hand
- **Projects graded**
  - View feedback on gitlab

ASSORTED MINUTIAE

- **Exams graded**
  - Should have in hand
- **Projects graded**
  - View feedback on gitlab
- **P2 checkpoint today**

ASSORTED MINUTIAE

- **Exams graded**
  - Should have in hand
- **Projects graded**
  - View feedback on gitlab
- **P2 checkpoint today**
  - Also a good opportunity to discuss the edam with me

ASSORTED MINUTIAE

- **Exams graded**
  - Should have in hand
- **Projects graded**
  - View feedback on gitlab
- **P2 checkpoint today**
  - Also a good opportunity to discuss the edam with me
- **New exercise out tonight**

MIDTERM RECAP

- **Overall, scores were good on this exam**

MIDTERM RECAP

- **Overall, scores were good on this exam**
  - A couple mistakes in missing the problem

MIDTERM RECAP

- **Overall, scores were good on this exam**
  - A couple mistakes in missing the problem
  - Lack of thoroughness in design decision problem --- no solution for how to getTopWords()

MIDTERM RECAP

- **Overall, scores were good on this exam**
  - A couple mistakes in missing the problem
  - Lack of thoroughness in design decision problem --- no solution for how to getTopWords()
  - Available today in office hours for regrades

MIDTERM RECAP

- **Overall, scores were good on this exam**
  - A couple mistakes in missing the problem
  - Lack of thoroughness in design decision problem --- no solution for how to getTopWords()
  - Available today in office hours for regrades
  - Regrading also available after class Friday

SORTING

- **Problem statement:**
  - Collection of Comparable data

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

- **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:

SORTING

- **Important definitions**
  - In-place: Requires only O(1) extra memory

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

- **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.

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:

SORTING

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

SORTING

- **Important definitions**
  - Interruptable: 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: 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: 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? 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, do not replace with an element that ties.

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, do not replace with an element that ties.
    - 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, do not replace with an element that ties.
    - 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? Same as before, if we maintain sorted order in case of ties.

SORTING

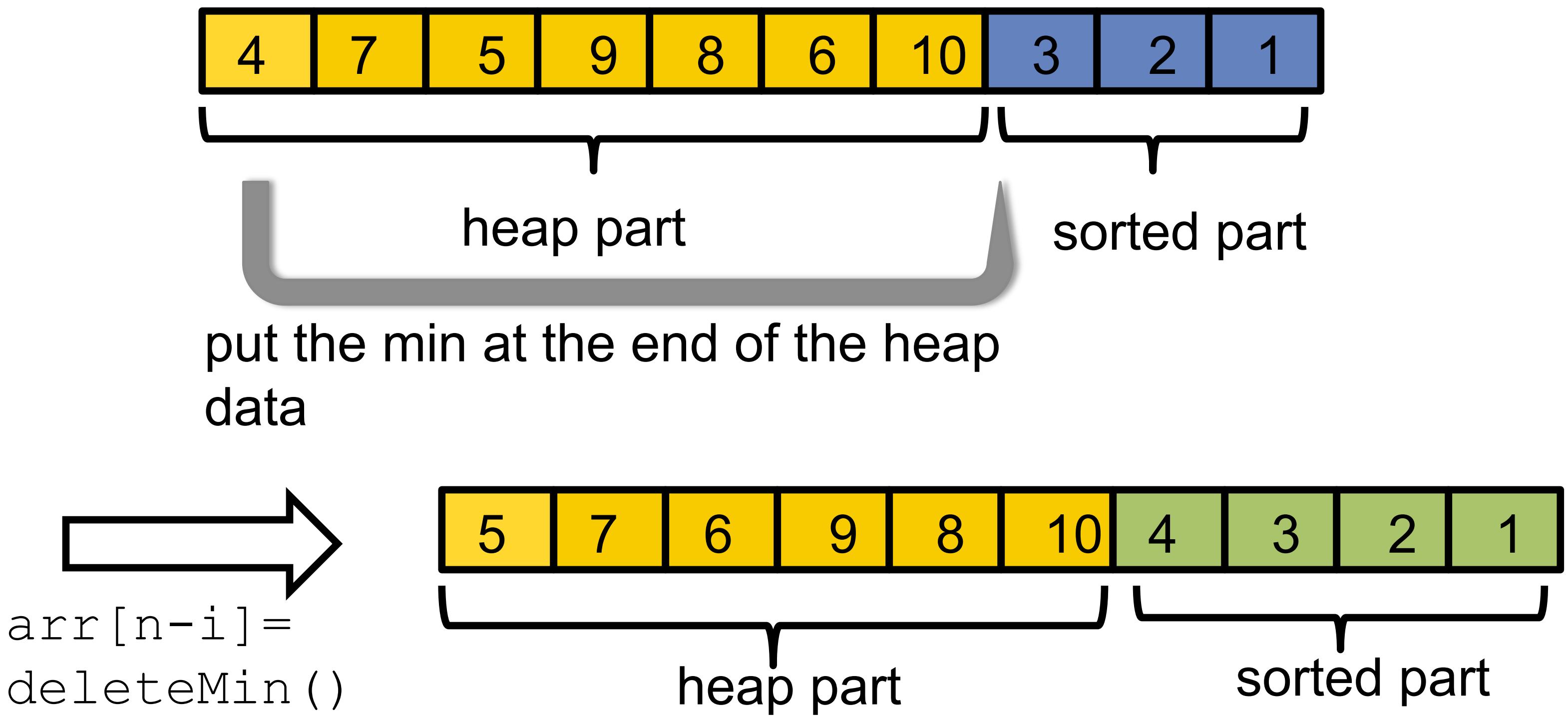
- **What are the sorts we've seen so far?**
  - Insertion Sort:
    - Algorithm? Maintain a sorted portion at the beginning of the array. For each new element, we swap it into the sorted portion until it reaches it's correct location
    - Runtime?
      - Worst-case: $O(n^2)$ – reverse sorted order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

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!



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:

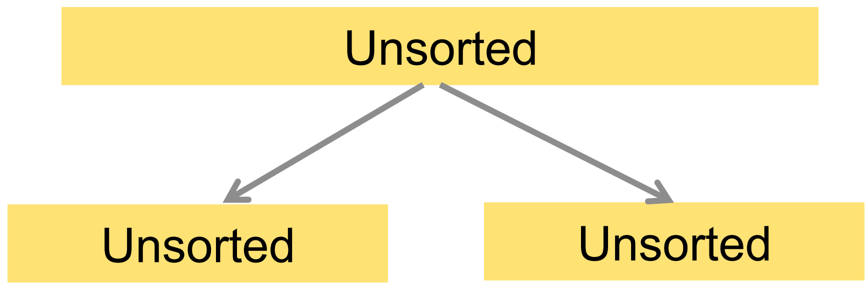
Sort the left half of the elements (recursively)
Sort the right half of the elements (recursively)
Merge the two sorted halves into a sorted whole

Quicksort:

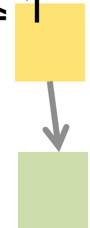
Pick a "pivot" element
Divide elements into less-than pivot and greater-than pivot
Sort the two divisions (recursively on each)
Answer is: sorted-less-than....pivot....sorted-greater-than

MERGE SORT

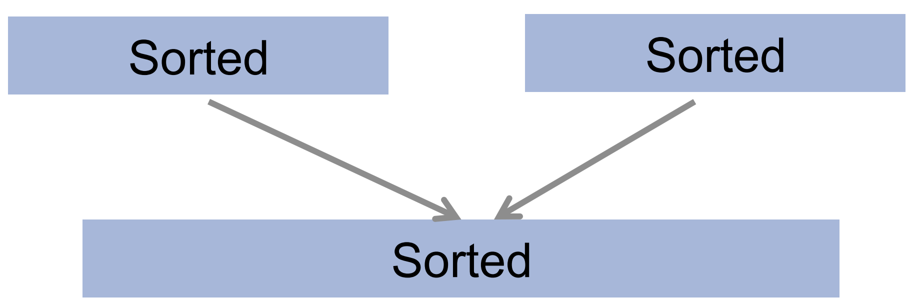
Divide: Split array roughly into half



Conquer: Return array when length ≤ 1



Combine: Combine two sorted arrays using merge

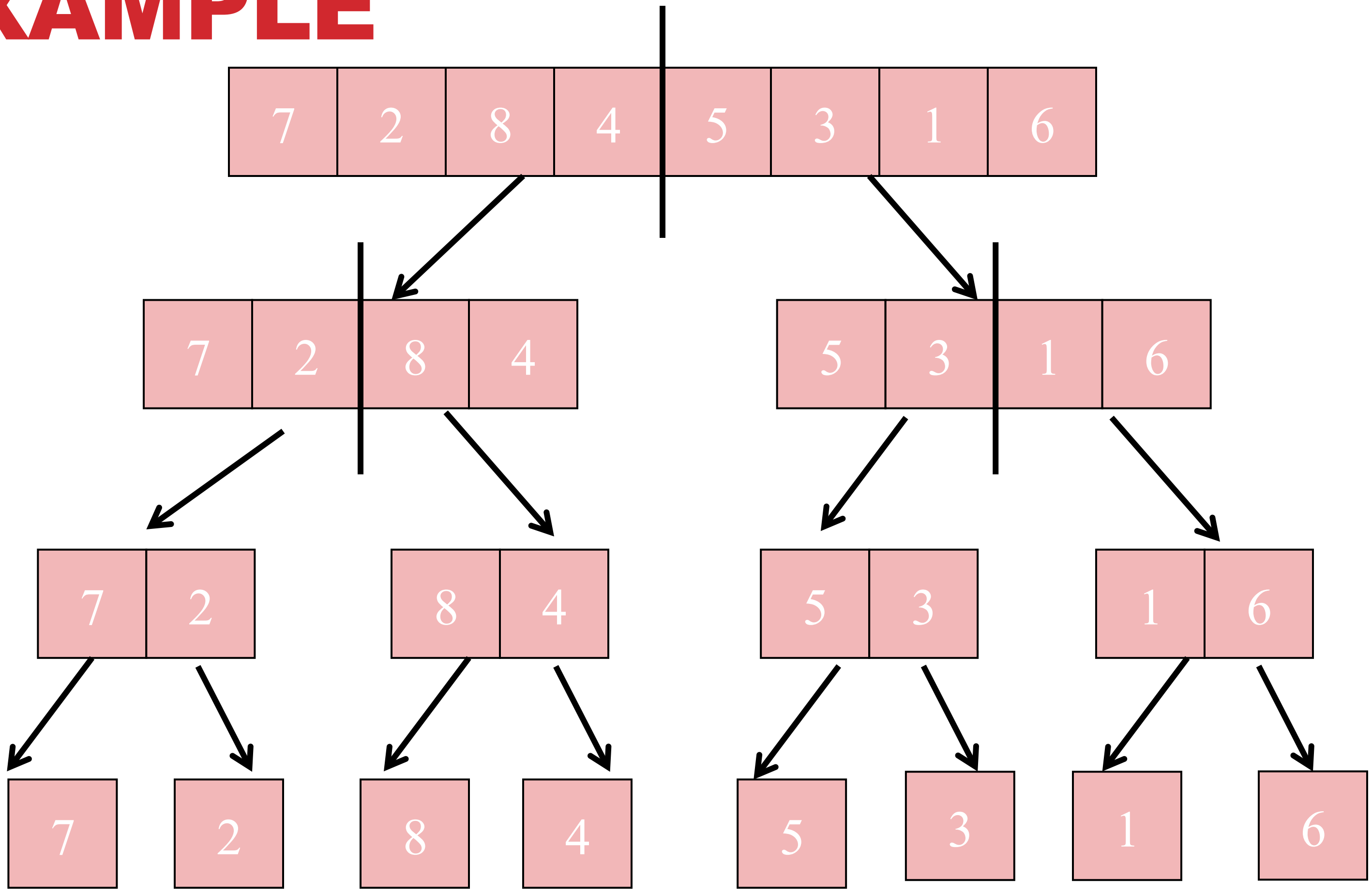


MERGE SORT: PSEUDOCODE

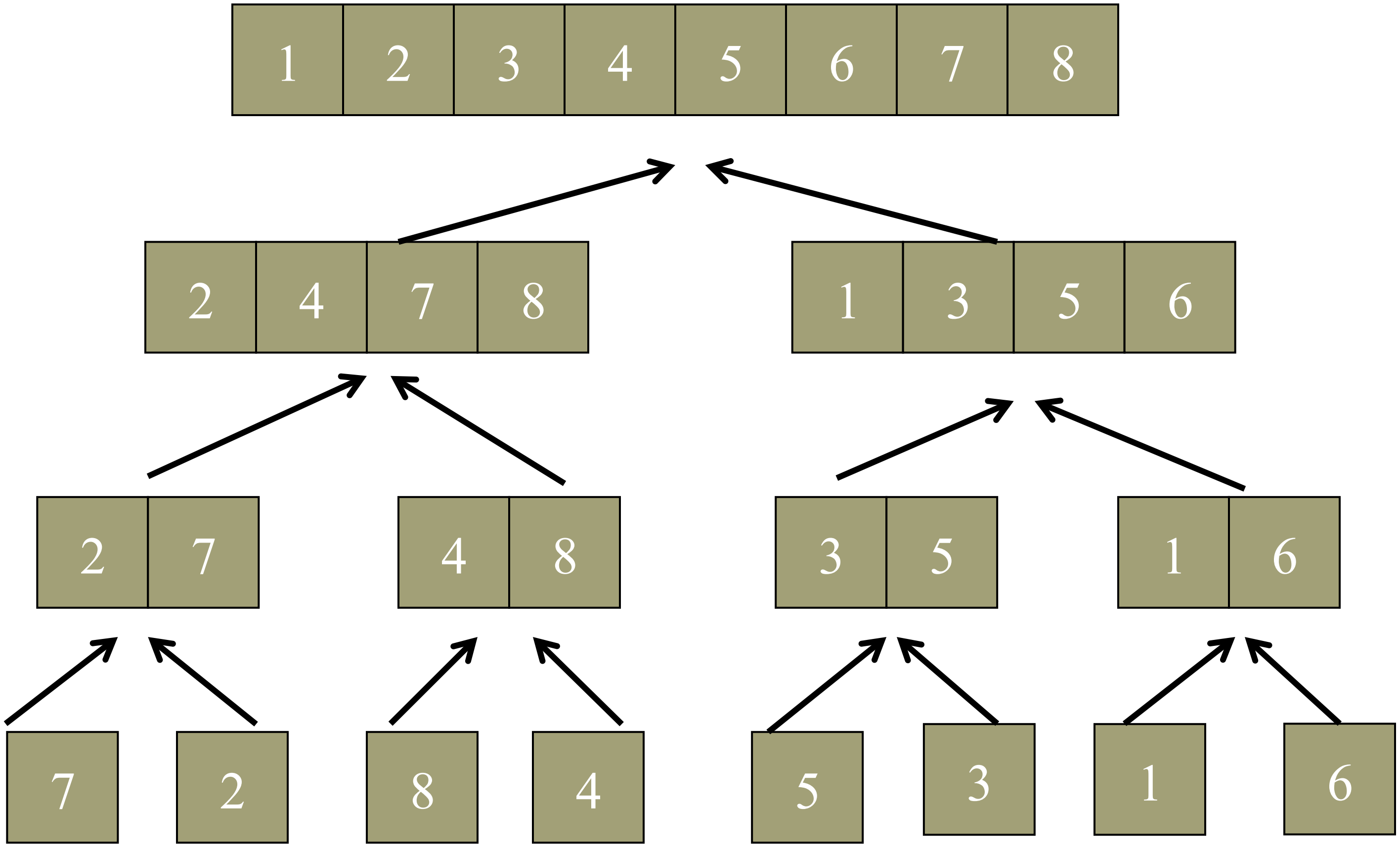
Core idea: split array in half, sort each half, merge back together. If the array has size 0 or 1, just return it unchanged

```
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

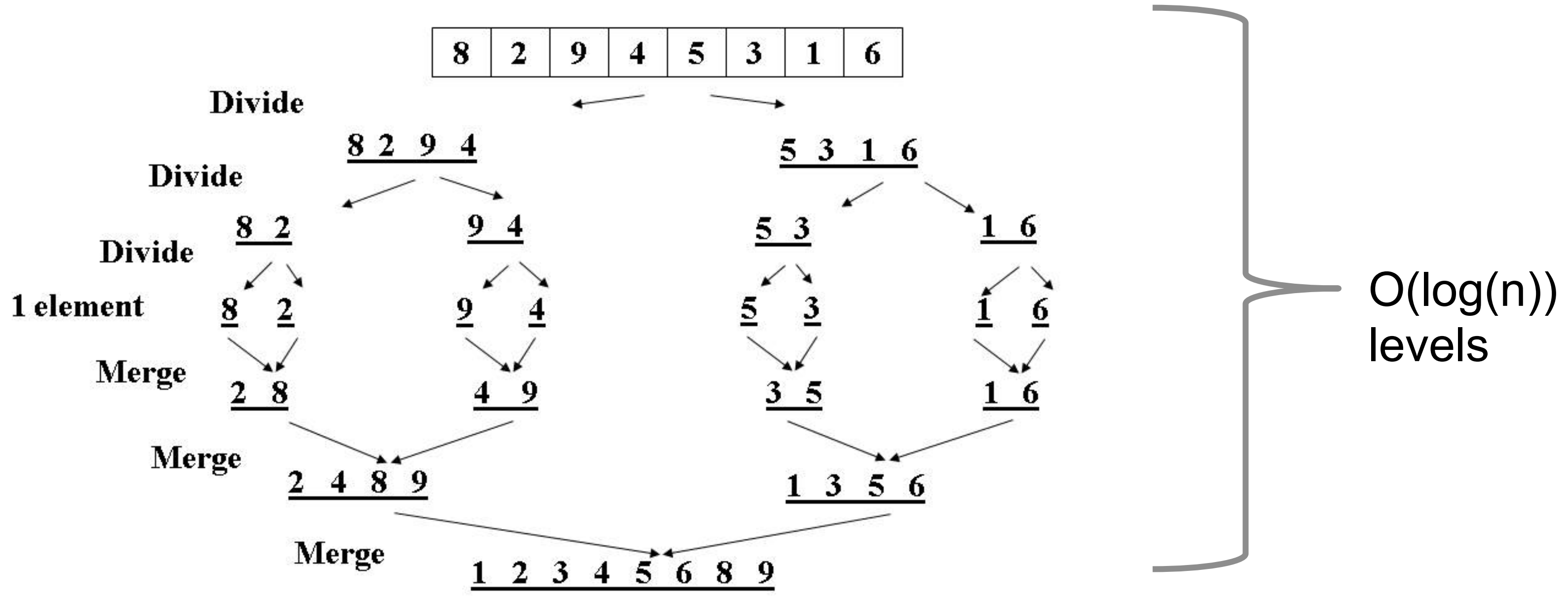


MERGE SORT ANALYSIS

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))



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.

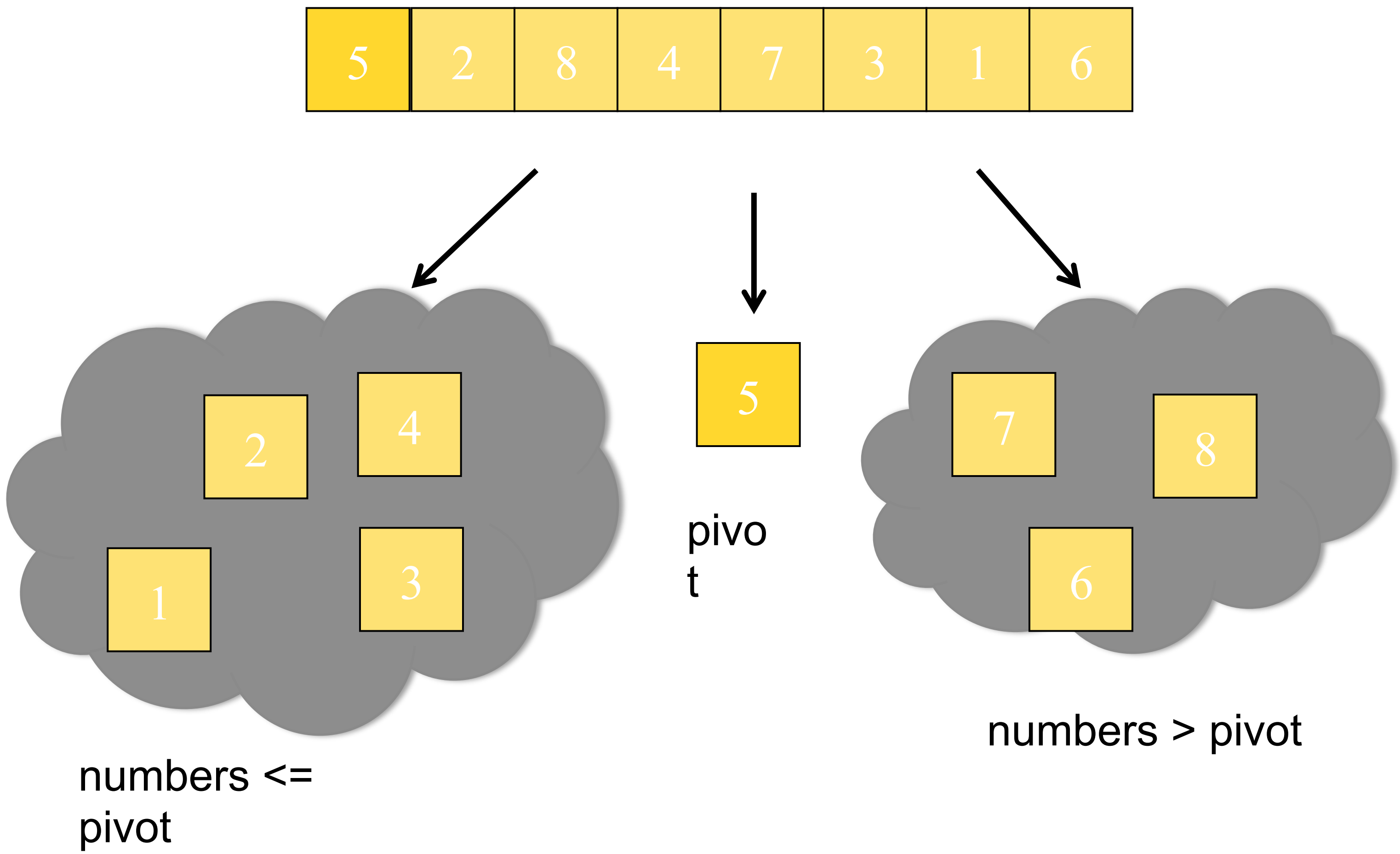
We're constantly copying and creating new arrays at each level...

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.

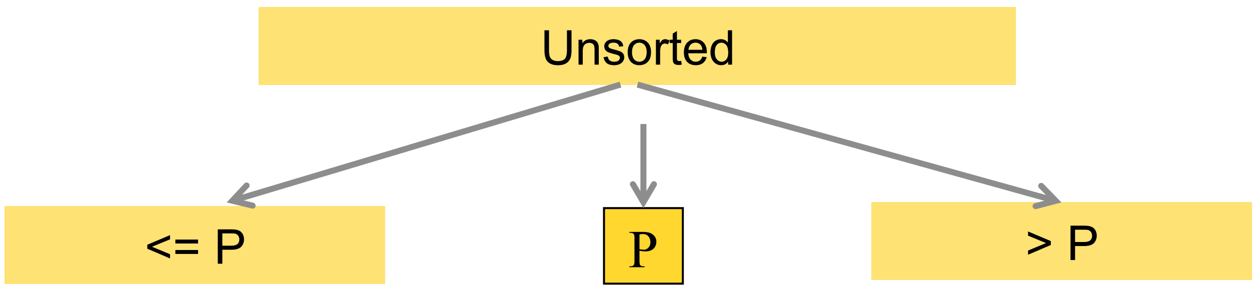
QUICK SORT

Divide: Split array around a 'pivot'

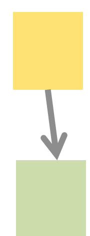


QUICK SORT

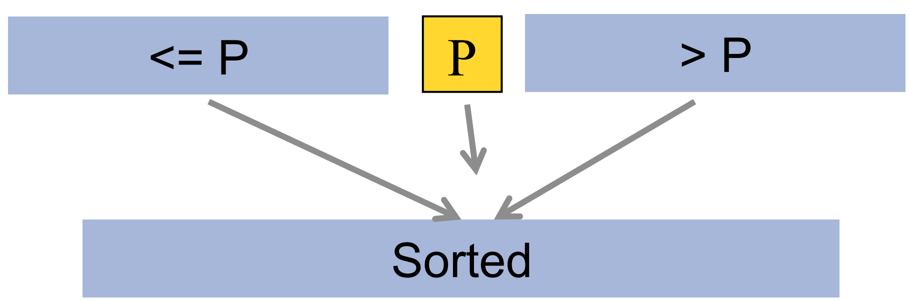
Divide: Pick a pivot, partition into groups



Conquer: Return array when length ≤ 1



Combine: Combine sorted partitions and pivot

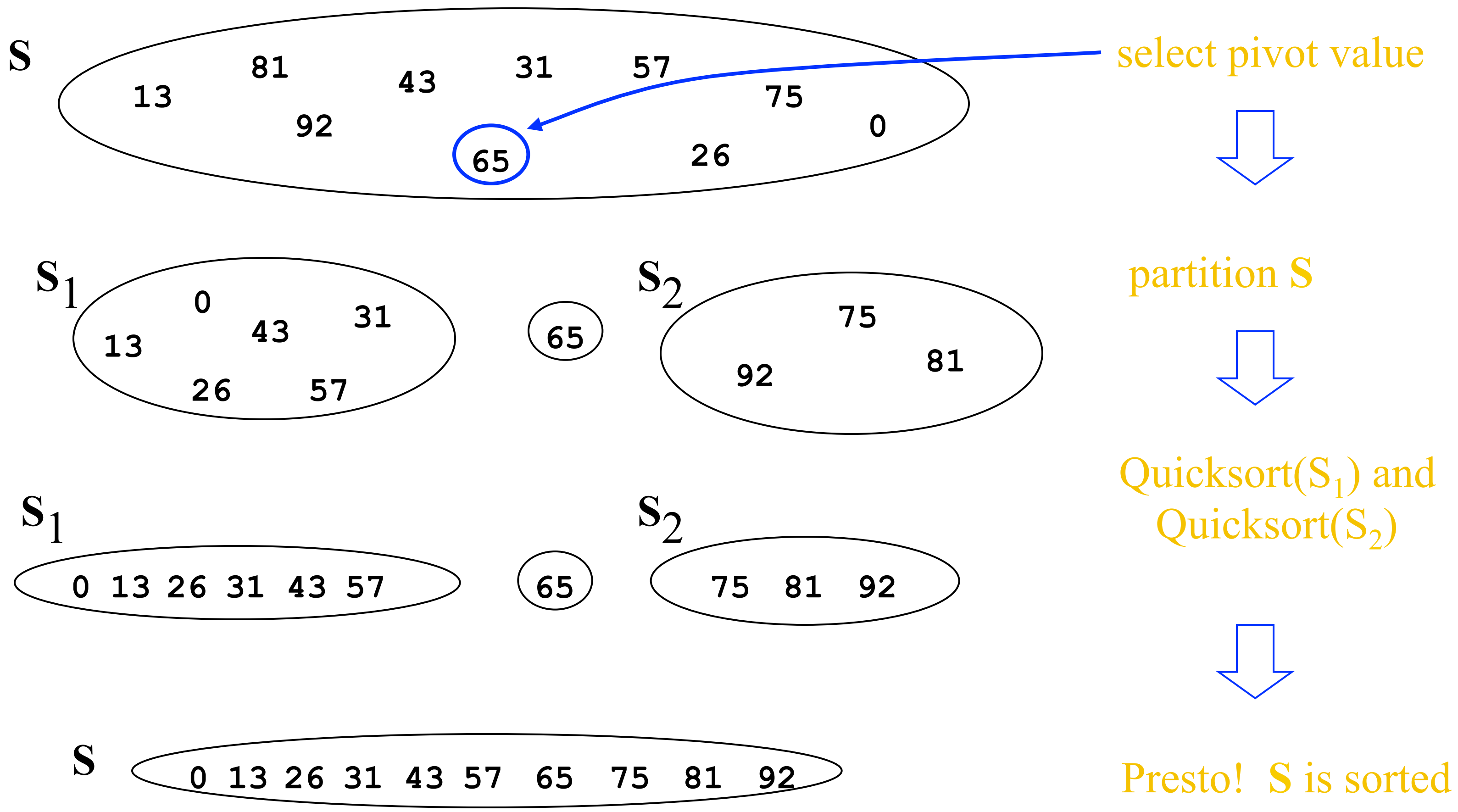


QUICK SORT PSEUDOCODE

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.

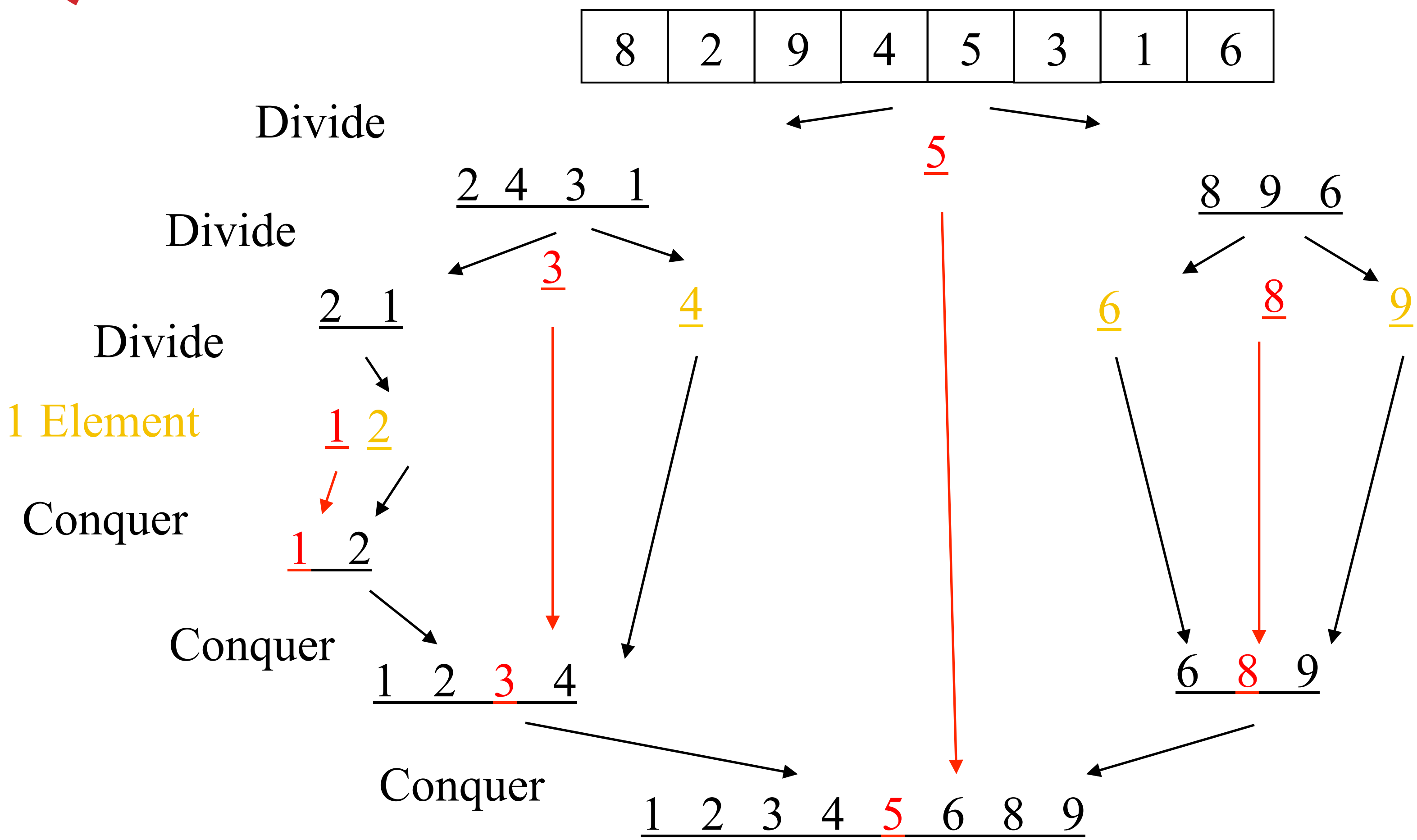
```
quicksort(input) {
    if (input.length < 2) {
        return input;
    } else {
        pivot = getPivot(input);
        smallerHalf = sort(getSmaller(pivot, input));
        largerHalf = sort(getBigger(pivot, input));
        return smallerHalf + pivot + largerHalf;
    }
}
```

QUICKSORT



[Weiss]

QUICKSORT



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

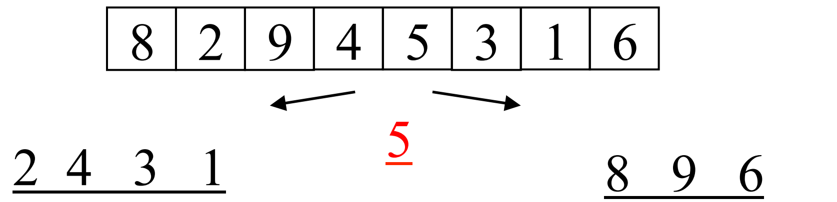
How to implement partitioning

- In linear time
- In place

PIVOTS

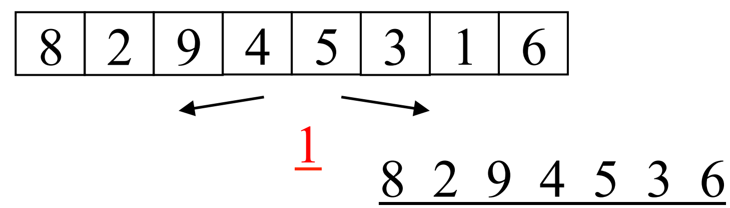
Best pivot?

- 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**