CSE 373

MAY 22ND – EVEN MORE SORTING

ASSORTED MINUTIAE

- **HW6 out tonight – Due next Tuesday at midnight**

ASSORTED MINUTIAE

- HW6 out tonight – Due next Tuesday at midnight
- Extra assignment – Due next Friday, last day of class

ASSORTED MINUTIAE

- **HW6 out tonight – Due next Tuesday at midnight**
- **Extra assignment – Due next Friday, last day of class**
    - No late days for this one

REVIEW

- Slow sorts

REVIEW

- **Slow sorts**
  - $O(n^2)$

REVIEW

- **Slow sorts**
  - $O(n^2)$
  - Insertion

REVIEW

- **Slow sorts**
  - $O(n^2)$
  - Insertion
  - Selection

REVIEW

- **Slow sorts**
  - $O(n^2)$
  - Insertion
  - Selection
- **Fast sorts**

REVIEW

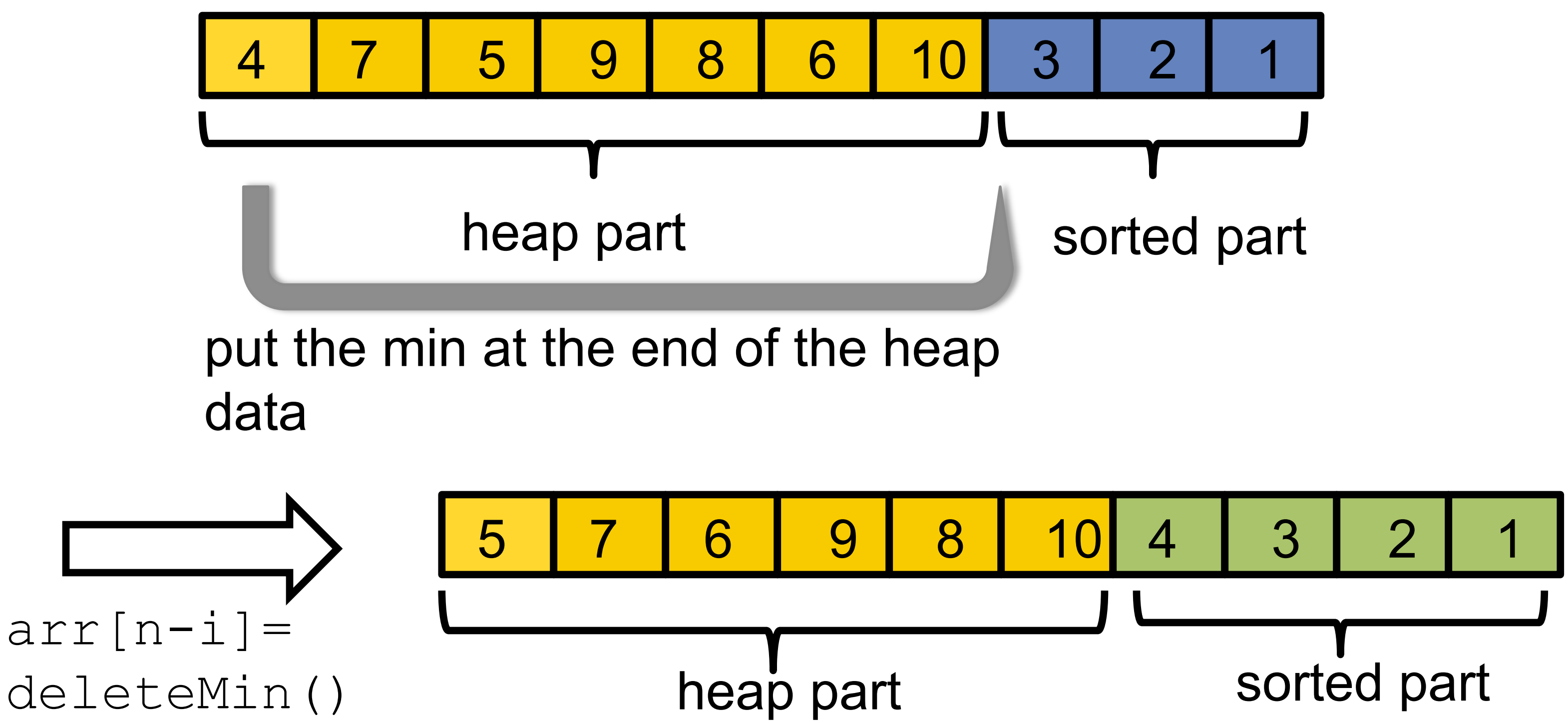
- **Slow sorts**
  - $O(n^2)$
  - Insertion
  - Selection
- **Fast sorts**
  - $O(n \log n)$

REVIEW

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

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!



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: Ω(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:

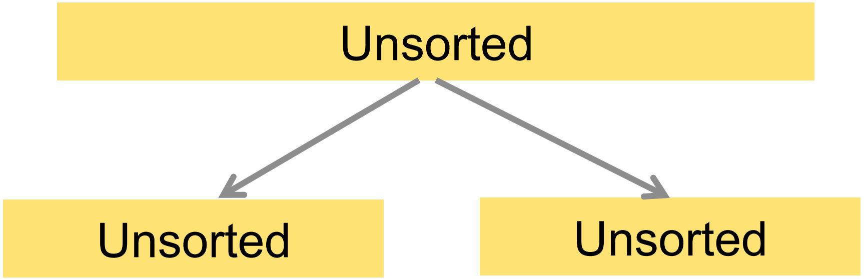
Sort the left half of the elements (recursively)
Sort the right half of the elements (recursively)
Merge the two sorted halves into a sorted whole

Quicksort:

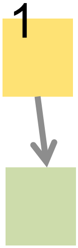
Pick a “pivot” element
Divide elements into less-than pivot and greater-than pivot
Sort the two divisions (recursively on each)
Answer is: sorted-less-than....pivot....sorted-greater-than

MERGE SORT

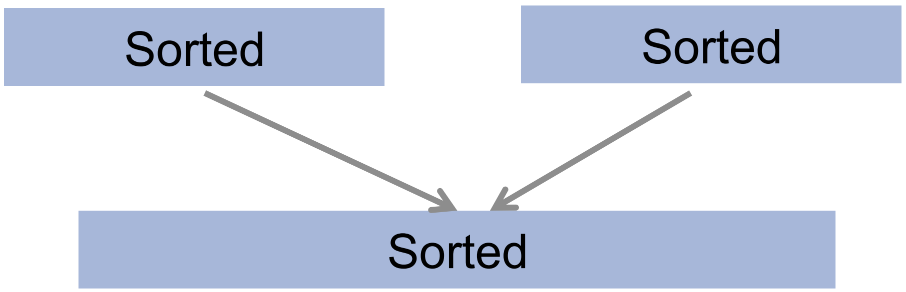
Divide: Split array roughly into half



Conquer: Return array when length ≤ 1



Combine: Combine two sorted arrays using merge

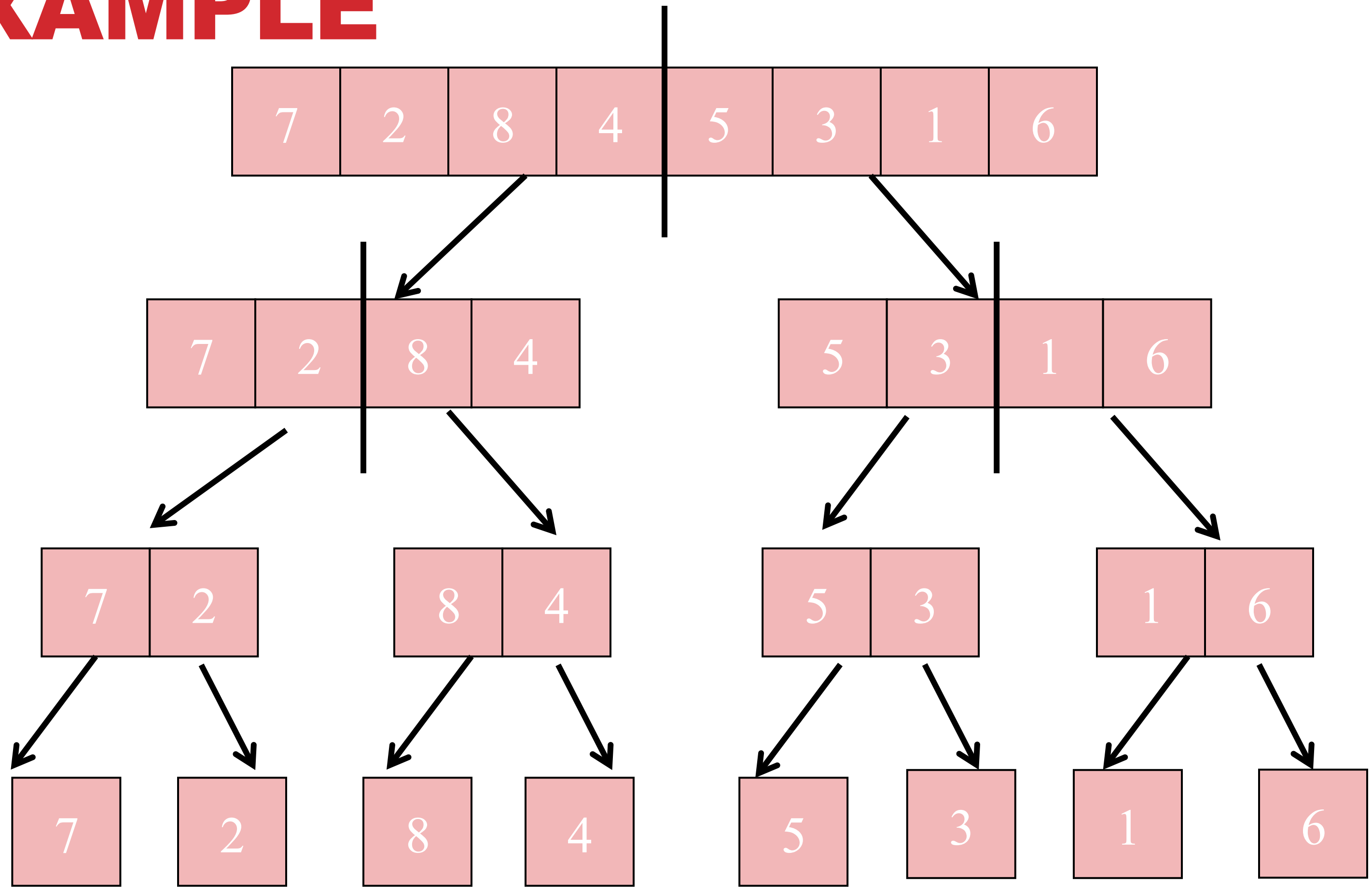


MERGE SORT: PSEUDOCODE

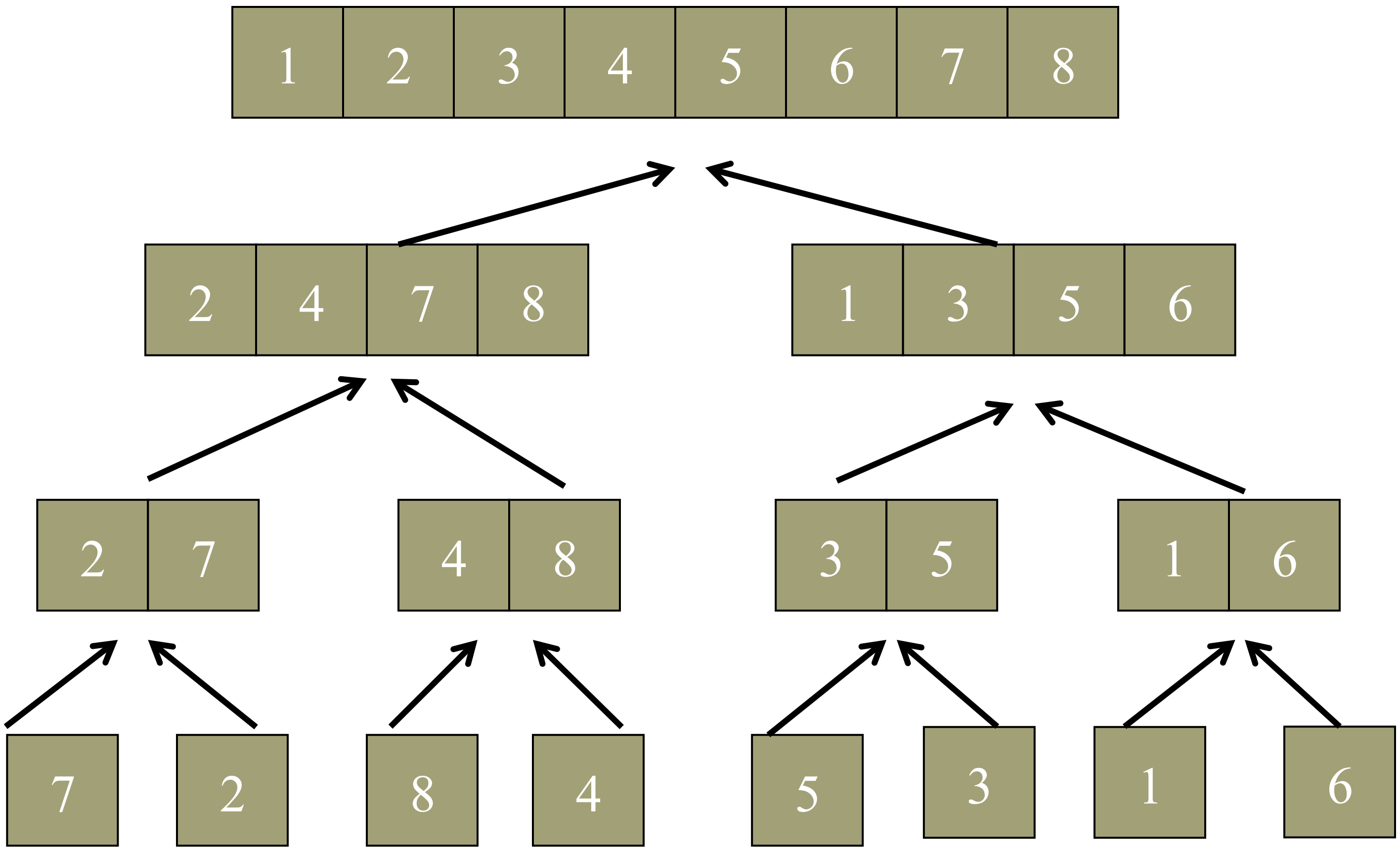
Core idea: split array in half, sort each half, merge back together. If the array has size 0 or 1, just return it unchanged

```
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

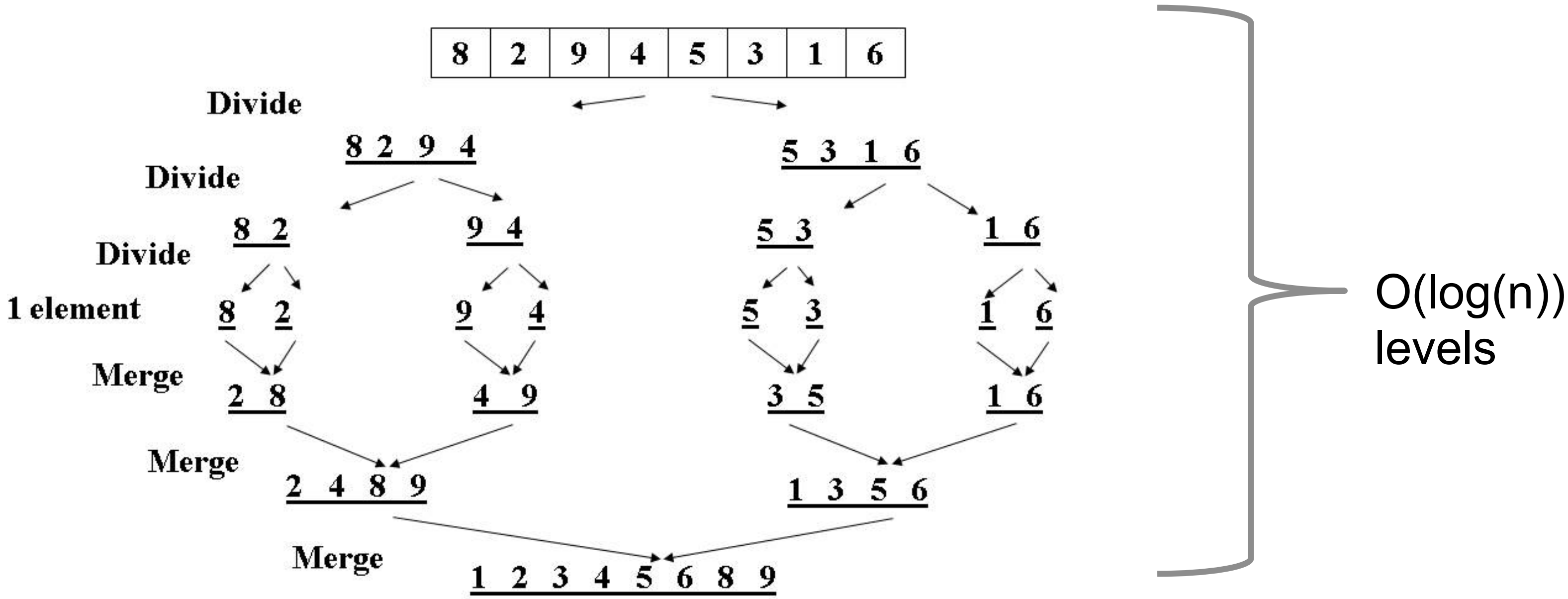


MERGE SORT ANALYSIS

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))



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.

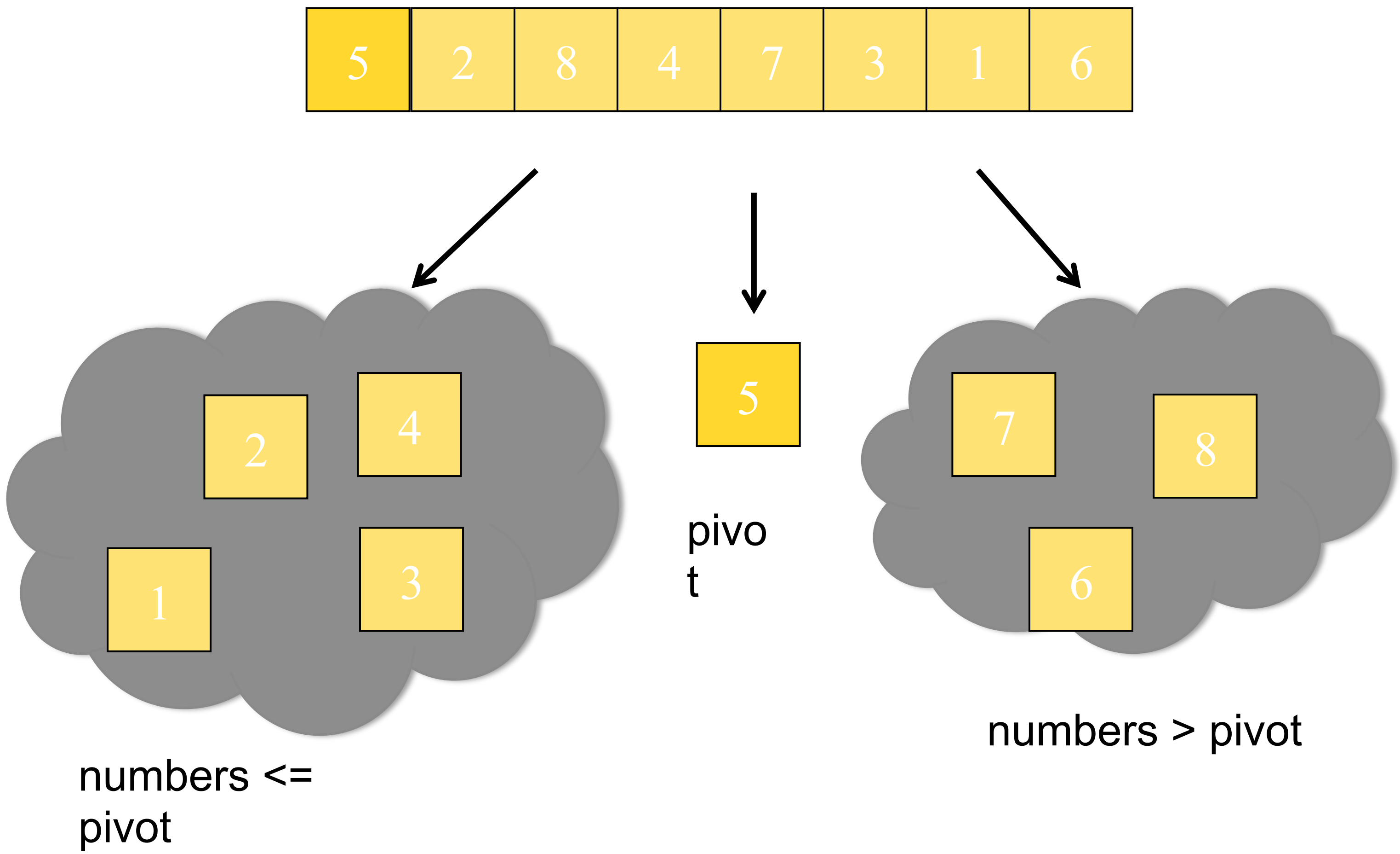
We're constantly copying and creating new arrays at each level...

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.

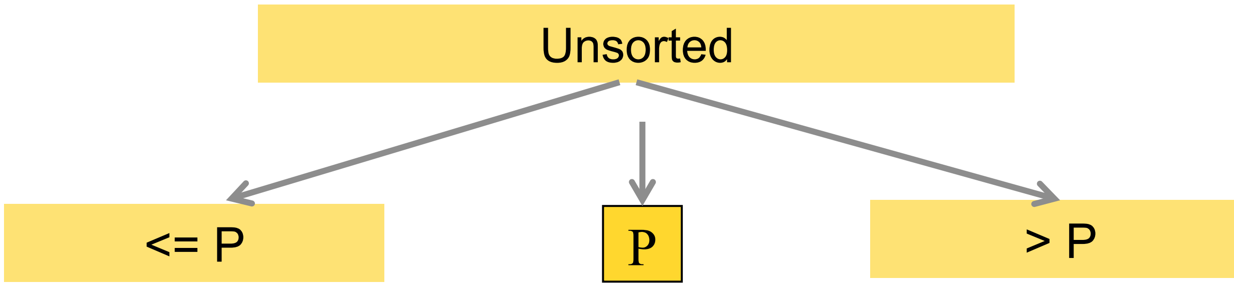
QUICK SORT

Divide: Split array around a ‘pivot’

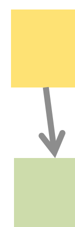


QUICK SORT

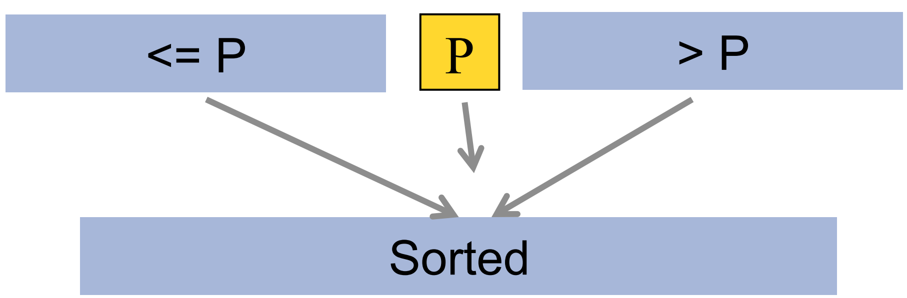
Divide: Pick a pivot, partition into groups



Conquer: Return array when length ≤ 1



Combine: Combine sorted partitions and pivot

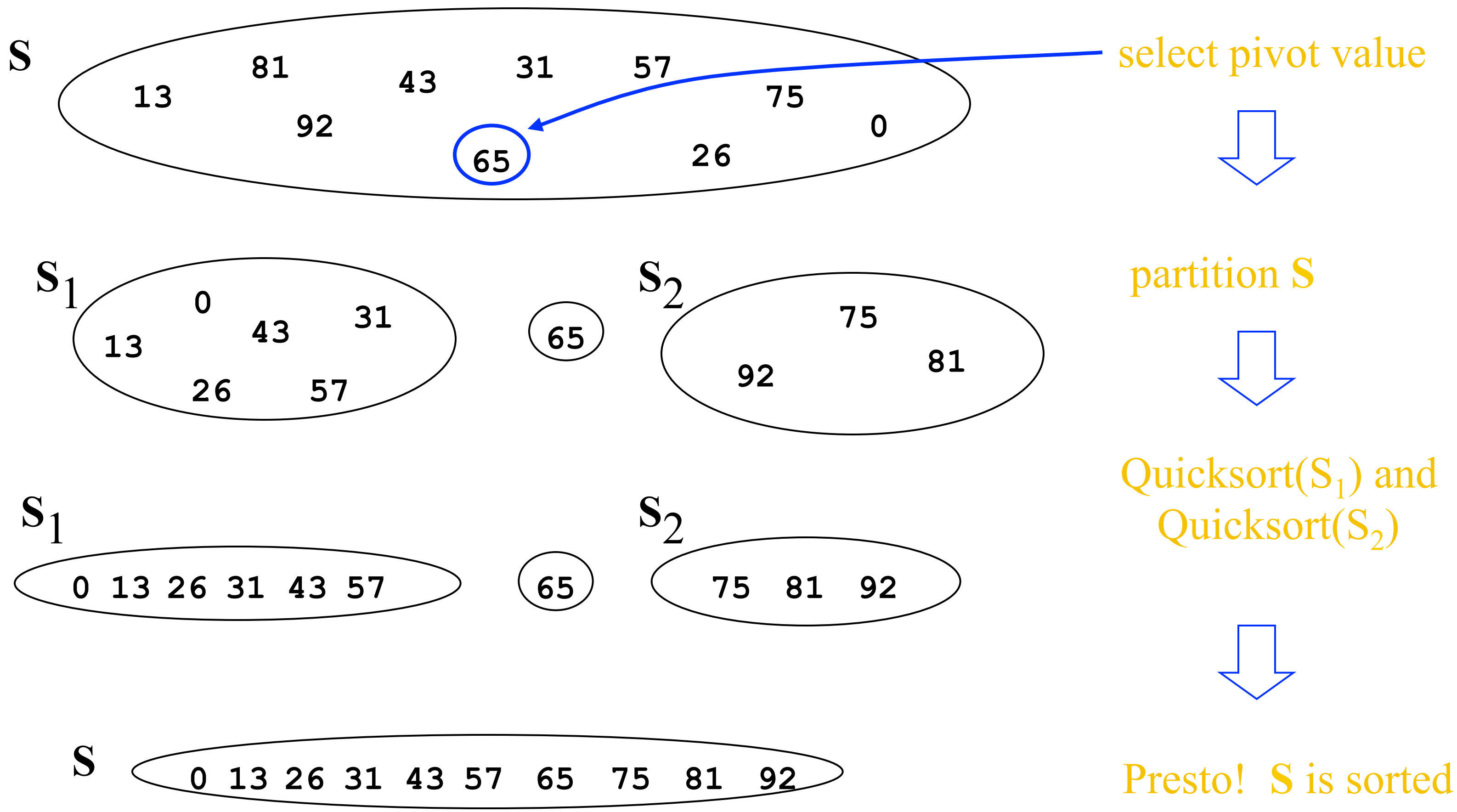


QUICK SORT PSEUDOCODE

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.

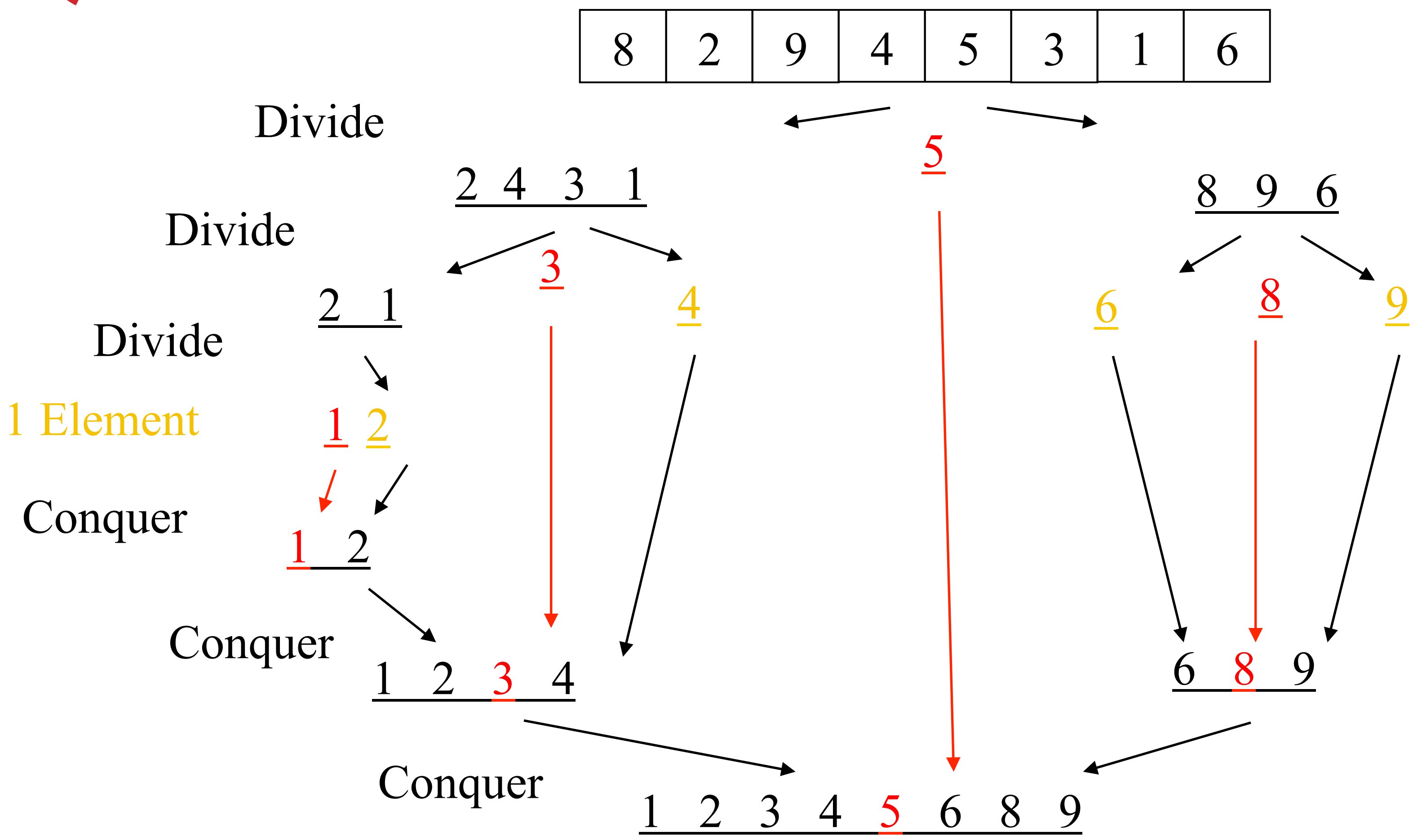
```
quicksort(input) {
    if (input.length < 2) {
        return input;
    } else {
        pivot = getPivot(input);
        smallerHalf = sort(getSmaller(pivot, input));
        largerHalf = sort(getBigger(pivot, input));
        return smallerHalf + pivot + largerHalf;
    }
}
```

QUICKSORT



[Weiss]

QUICKSORT



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

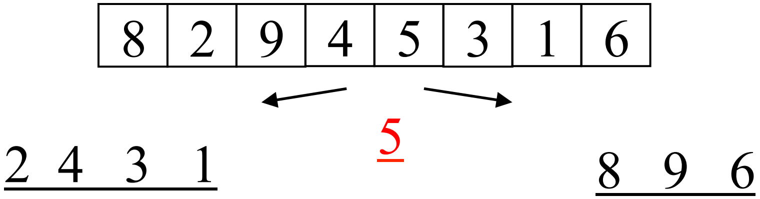
- Any choice is correct: data will end up sorted
- But as analysis will show, want the two partitions to be about equal in size

How to implement partitioning

- In linear time
- In place

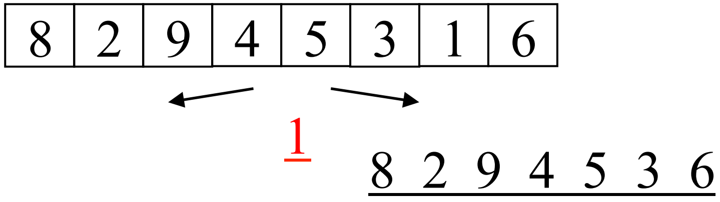
PIVOTS

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**

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

ASYMPTOTIC RUNTIME OF RECURSION

Recurrence Definition:

A recurrence is a recursive definition of a function in terms of smaller values.

Example: Fibonacci numbers.

To analyze the runtime of recursive code, we use a recurrence by splitting the work into two pieces:

- Non-Recursive Work
- Recursive Work

RECURSIVE VERSION OF SUM:

```
int sum(int[] arr){
   return help(arr,0,arr.length);
}
int help(int[] arr, int lo, int hi) {
   if(lo==hi)   return 0;
   if(lo==hi-1) return arr[lo];
   int mid = (hi+lo)/2;
   return help(arr,lo,mid) + help(arr,mid,hi);
}
```

What's the recurrence T(n)?

- Non-Recursive Work: O(1)
- Recursive Work: T(n/2) * 2 halves

$T(n) = O(1) + 2*T(n/2)$

SOLVING THAT RECURRENCE RELATION

1. **Determine the recurrence relation. What is the base case?**
   - *If* $T(1) = 1$, then $T(n) = 1 + 2*T(n/2)$

2. **"Expand" the original relation to find an equivalent general expression *in terms of the number of expansions*.**
   - $T(n) = 1 + 2 * T(n / 2)$
     $= 1 + 2 + 2 * T(n / 4)$
     $= 1 + 2 + 4 + \ldots$ for log(n) times
     $= \ldots$
     $= 2^{(\log n)} - 1$

3. **Find a closed-form expression by setting *the number of expansions* to a value which reduces the problem to a base case**
   - So $T(n)$ is $O(n)$

Explanation: it adds each number once while doing little else

SOLVING RECURRENCE RELATIONS EXAMPLE 2

1. **Determine the recurrence relation. What is the base case?**
   - *If* $T(n) = 10 + T(n/2)$ and $T(1) = 10$

2. **"Expand" the original relation to find an equivalent general expression *in terms of the number of expansions*.**
   - $T(n) = 10 + 10 + T(n/4)$
     $= 10 + 10 + 10 + T(n/8)$
     $= \ldots$
     $= 10k + T(n/(2^k))$

3. **Find a closed-form expression by setting *the number of expansions* to a value which reduces the problem to a base case**
   - $n/(2^k) = 1$ means $n = 2^k$ means $k = \log_2 n$
   - So $T(n) = 10 \log_2 n + 8$ (get to base case and do it)
   - So $T(n)$ is $O(\log n)$

REALLY COMMON RECURRENCES

You can recognize some really common recurrences:

$T(n) = O(1) + T(n\text{-}1)$	linear
$T(n) = O(1) + 2T(n/2)$	linear
$T(n) = O(1) + T(n/2)$	logarithmic $O(\texttt{log}\ n)$
$T(n) = O(1) + 2T(n\text{-}1)$	exponential
$T(n) = O(n) + T(n\text{-}1)$	quadratic
$T(n) = O(n) + T(n/2)$	linear
$T(n) = O(n) + 2T(n/2)$	$O(n\ \texttt{log}\ n)$ (divide and conquer sort)

Note big-Oh can also use more than one variable

Example: can sum all elements of an *n*-by-*m* matrix in $O(nm)$

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