Lecture 21: Introduction to Sorting

CSE 373: Data Structures and Algorithms

Announcements

P4 due today

P5 out later today

Exercise 5 due on Friday

Exercise 6 out on Friday

Midterm 2 topics coming on Friday

Principle 3: Divide and Conquer

1. Divide your work into smaller pieces recursively
   - Pieces should be smaller versions of the larger problem

2. Conquer the individual pieces
   - Recursion!
   - Until you hit the base case

3. Combine the results of your recursive calls

```
divideAndConquer(input) {
   if (small enough to solve)
      conquer, solve, return results
   else
      divide input into a smaller pieces
      recurse on smaller piece
      combine results and return
}
```

Merge Sort

https://www.youtube.com/watch?v=XaqR3G_NVoo

Divide

0	1	2	3	4	5	6	7	8	9
8	2	91	22	57	1	10	6	7	4

0	1	2	3	4
8	2	91	22	57

5	6	7	8	9
1	10	6	7	4

Sort the pieces through the magic of recursion

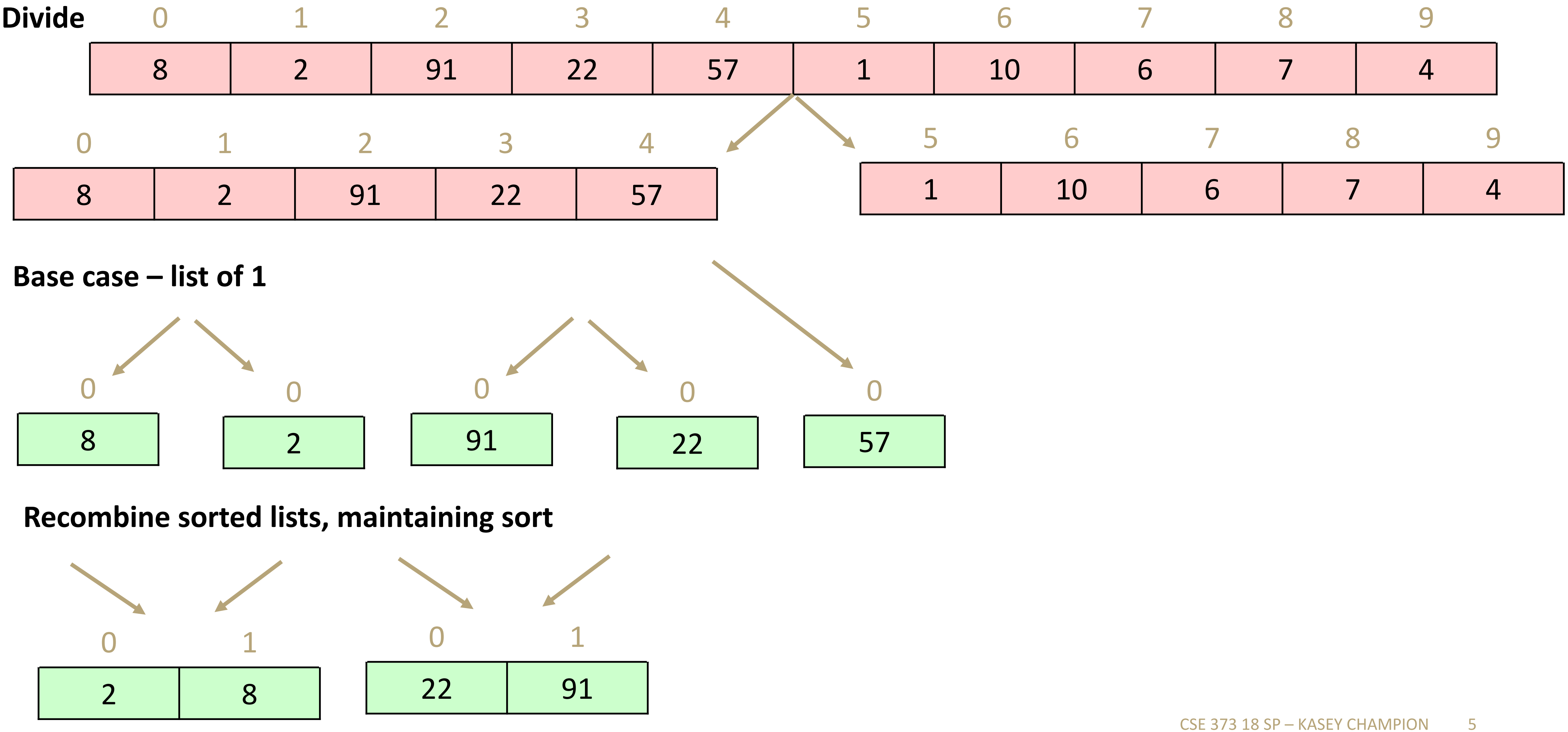
Combine

0	1	2	3	4
2	8	22	57	91

5	6	7	8	9
1	4	6	7	10

0	1	2	3	4	5	6	7	8	9
1	2	4	6	7	8	10	22	57	91

Merge Sort Divide



Divide

0	1	2	3	4	5	6	7	8	9
8	2	91	22	57	1	10	6	7	4

0	1	2	3	4
8	2	91	22	57

5	6	7	8	9
1	10	6	7	4

Base case – list of 1

0
8

0
2

0
91

0
22

0
57

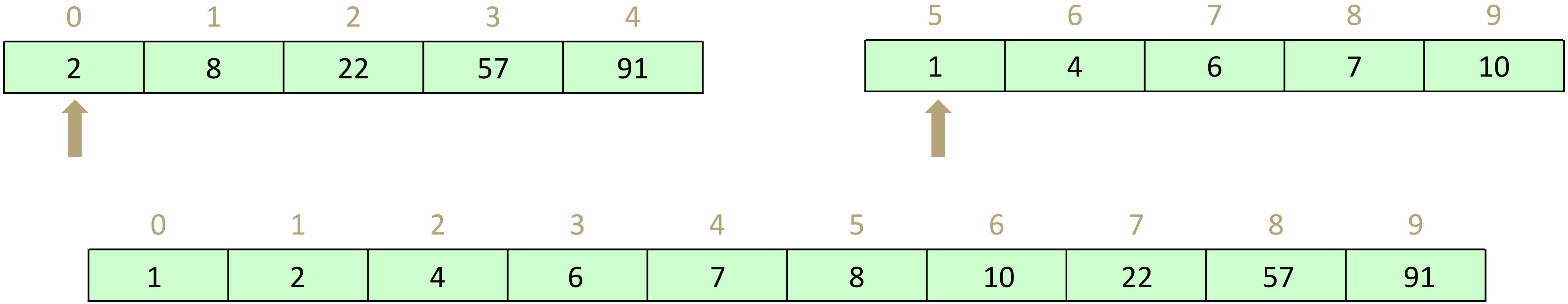
Recombine sorted lists, maintaining sort

0	1
2	8

0	1
22	91

Merge Sort Combine

Combine



0	1	2	3	4
2	8	22	57	91

5	6	7	8	9
1	4	6	7	10

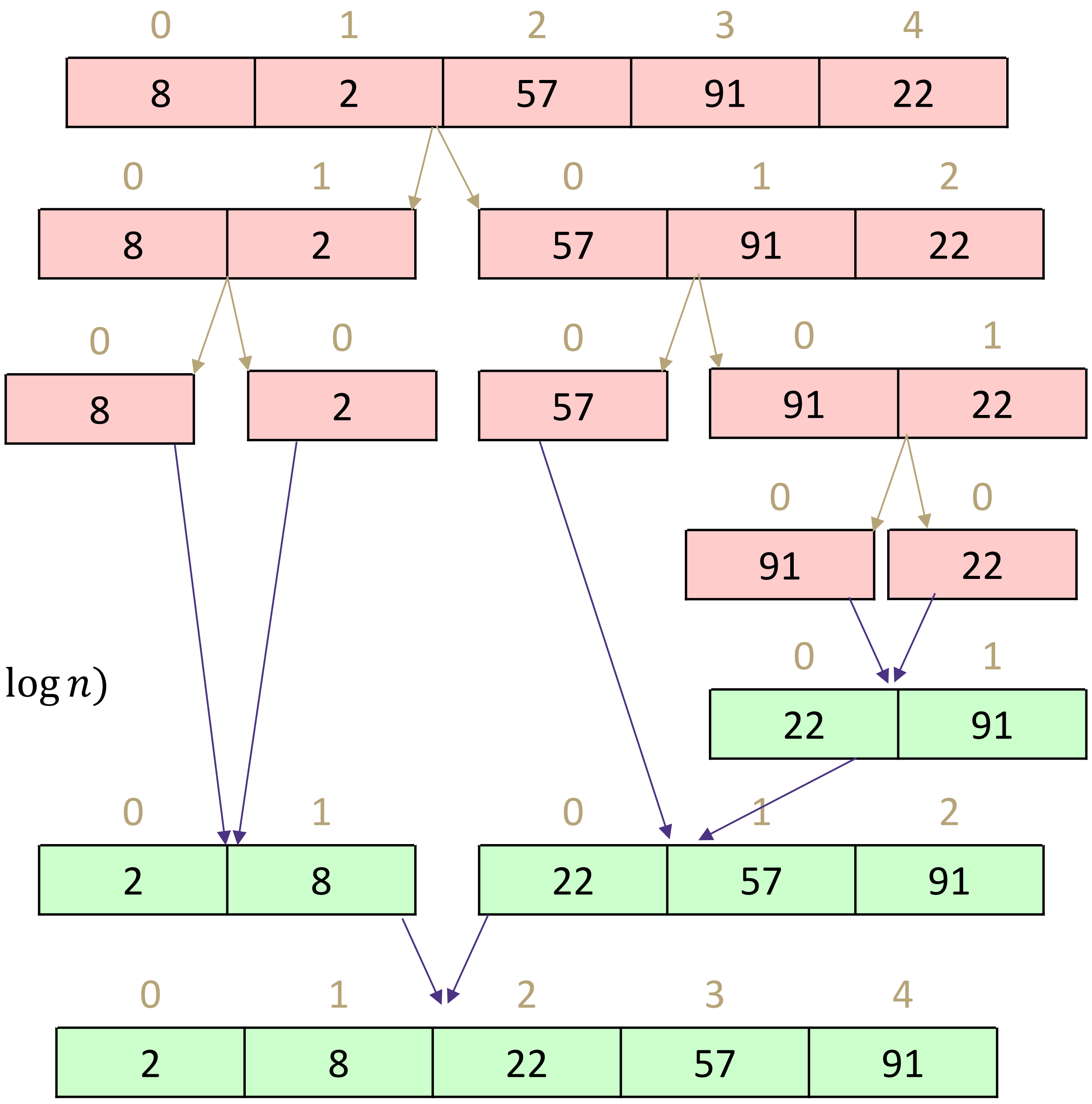
0	1	2	3	4	5	6	7	8	9
1	2	4	6	7	8	10	22	57	91

Recombination step compares two sorted arrays and combines them to maintain sorted order
- Starting at the front of each list two values are compared to decide which is smallest
- Smallest is added to combined array
- Pointer to “front” of smaller array who's value was just chosen is updated to consider next value
- Repeat until all values from smaller arrays are added to combined array

Merge Sort

```
mergeSort(input) {
   if (input.length == 1)
      return
   else
      smallerHalf = mergeSort(new [0, ..., mid])
      largerHalf = mergeSort(new [mid + 1, ...])
      return merge(smallerHalf, largerHalf)
}
```

Worst case runtime?	$T(n) = \begin{cases} 1 \text{ if } n<=1 \\ 2T(n/2) + n \text{ otherwise} \end{cases} = \Theta(n \log n)$
Best case runtime?	Same
In Practice runtime?	Same
Stable?	Yes
In-place?	No



Divide and Conquer

There’s more than one way to divide!

Mergesort:

Split into two arrays.

- Elements that just happened to be on the left and that happened to be on the right.

Quicksort:

Split into two arrays.

- Elements that are “small” and elements that are “large”
- What do I mean by “small” and “large” ?

Choose a “pivot” value (an element of the array)

One array has elements smaller than pivot, other has elements larger than pivot.

Quick Sort v1

https://www.youtube.com/watch?v=ywWBy6J5gz8

Divide

0	1	2	3	4	5	6	7	8	9
8	2	91	22	57	1	10	6	7	4

0	1	2	3	4
2	1	6	7	4

0
8

0	1	2	3
91	22	57	10

Sort the pieces through the magic of recursion

0	1	2	3	4
1	2	4	6	7

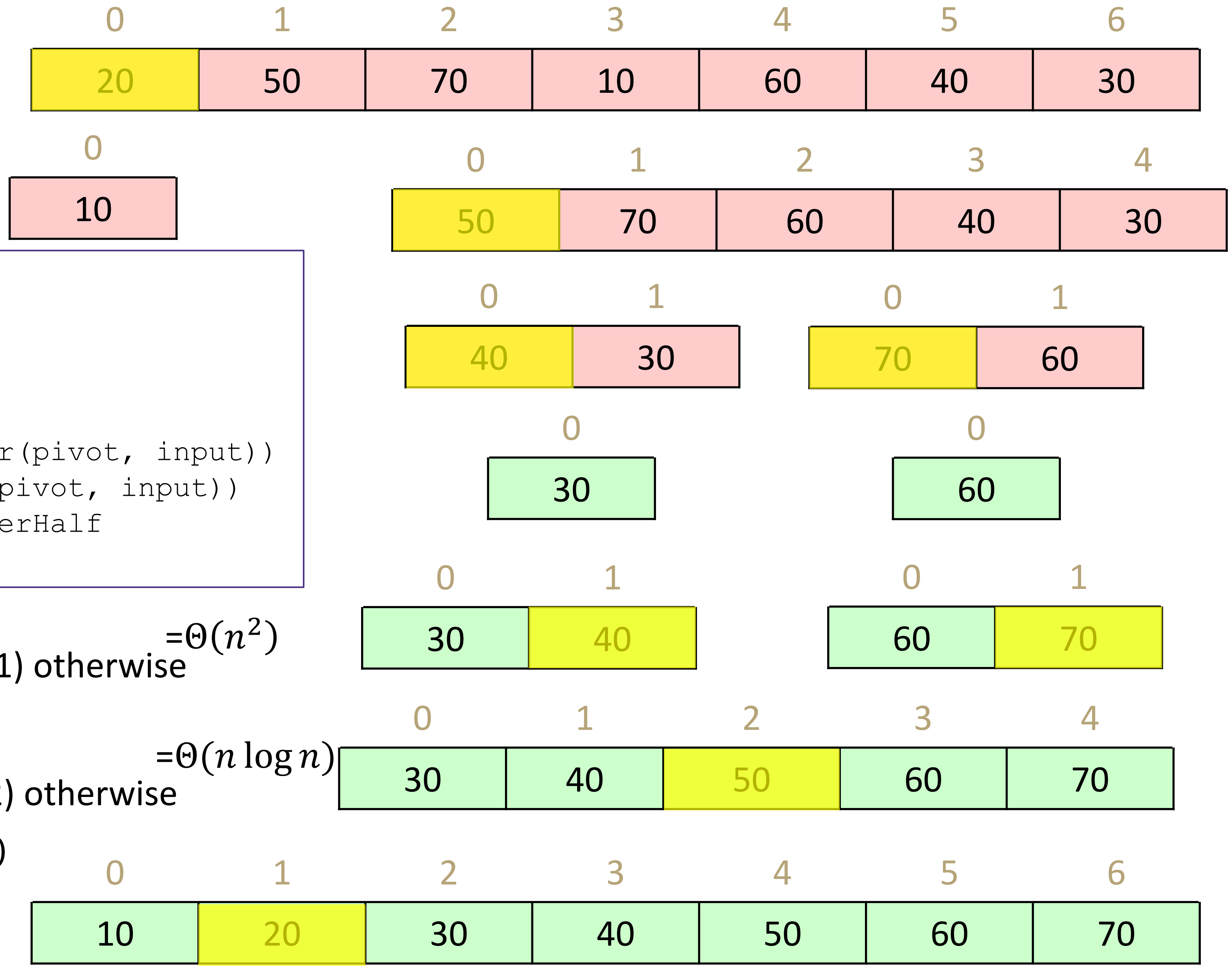
0
8

0	1	2	3
10	22	57	91

Combine (no extra work if in-place)

0	1	2	3	4	5	6	7	8	9
1	2	4	6	7	8	10	22	57	91

Quick Sort v1



0	1	2	3	4	5	6
20	50	70	10	60	40	30

0
10

0	1	2	3	4
50	70	60	40	30

0	1
40	30

0	1
70	60

0
30

0
60

0	1
30	40

0	1
60	70

0	1	2	3	4
30	40	50	60	70

0	1	2	3	4	5	6
10	20	30	40	50	60	70

```
quickSort(input) {
   if (input.length == 1)
      return
   else
      pivot = getPivot(input)
      smallerHalf = quickSort(getSmaller(pivot, input))
      largerHalf = quickSort(getBigger(pivot, input))
      return smallerHalf + pivot + largerHalf
}
```

Worst case runtime? $T(n) = \begin{cases} 1 \text{ if } n<= 1 \\ n + T(n - 1) \text{ otherwise} \end{cases}$ $=\Theta(n^2)$

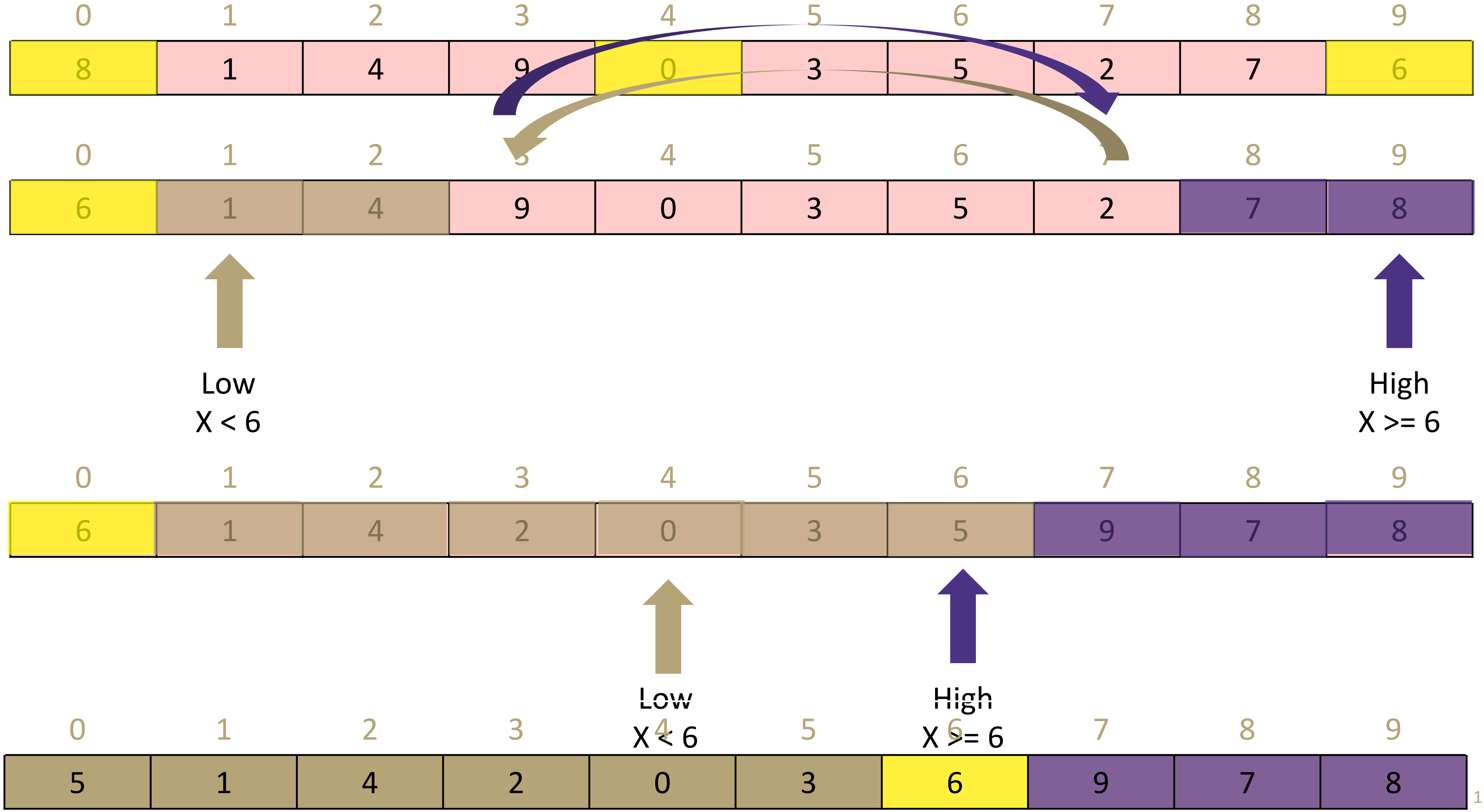
Best case runtime? $T(n) = \begin{cases} 1 \text{ if } n<= 1 \\ n + 2T(n/2) \text{ otherwise} \end{cases}$ $=\Theta(n \log n)$

In-practice runtime? Just trust me $\Theta(n \log n)$

Stable? No

In-place? Can be done

Quick Sort v2 (in-place)



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

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

Low
X < 6

High
X >= 6

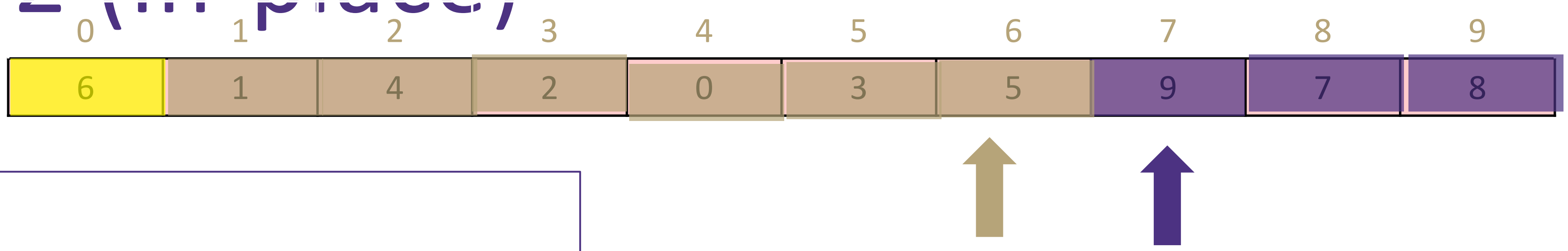
0	1	2	3	4	5	6	7	8	9
6	1	4	2	0	3	5	9	7	8

Low
X < 6

High
X >= 6

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

Quick Sort v2 (in-place)



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

```
quickSort(input) {
   if (input.length == 1)
      return
   else
      pivot = getPivot(input)
      smallerHalf = quickSort(getSmaller(pivot, input))
      largerHalf = quickSort(getBigger(pivot, input))
      return smallerHalf + pivot + largerHalf
}
```

Worst case runtime? $T(n) = \begin{cases} 1 \text{ if } n<= 1 \\ n + T(n - 1) \text{ otherwise} \end{cases} = \theta(n^2)$

Best case runtime? $T(n) = \begin{cases} 1 \text{ if } n<= 1 \\ n + 2T(n/2) \text{ otherwise} \end{cases} = \theta(n \log n)$

In-practice runtime? $= \theta(n \log n)$ Just trust me

Stable? No

In-place? Yes

Can we do better?

We’d really like to avoid hitting the worst case.

Key to getting a good running time, is always cutting the array (about) in half.

How do we choose a good pivot?

Here are four options for finding a pivot. What are the tradeoffs?

- Just take the first element
- Take the median of the first, last, and middle element
- Take the median of the full array
- Pick a random element as a pivot

Pivots

Just take the first element

- fast to find a pivot
- But (e.g.) nearly sorted lists get $\Omega(n^2)$ behavior overall

Take the median of the first, last, and middle element

- Guaranteed to not have the absolute smallest value.
- On real data, this works quite well…
- But worst case is still $\Omega(n^2)$

Median of three is a common choice in practice

Take the median of the full array

- Can actually find the median in $O(n)$ time (google QuickSelect). It's **complicated.**
- $O(n\ log\ n)$ even in the worst case….but the constant factors are **awful**. No one does quicksort this way.

Pick a random element as a pivot

- somewhat slow constant factors
- Get $O(n \log n)$ running time with probability at least $1 - 1/n^2$
- "adversaries" can't make it more likely that we hit the worst case.

Summary

	Best-Case	Worst-Case	Space / Memory	Stable
Selection Sort	Θ(n^2)	Θ(n^2)	Θ(1)	No
Insertion Sort	Θ(n)	Θ(n^2)	Θ(1)	Yes
Heap Sort	Θ(n)	Θ(nlogn)	Θ(n)	No
In-Place Heap Sort	Θ(n)	Θ(nlogn)	Θ(1)	No
Merge Sort	Θ(nlogn)	Θ(nlogn)	Θ(nlogn) Θ(n)* optimized	Yes
Quick Sort	Θ(nlogn)	Θ(n^2)	Θ(n)	No
In-place quick sort	Θ(nlogn)	Θ(n^2)	Θ(1)	No

What does Java do?

For Objects – merge sort

For primitives – Dual Pivot Quick Sort

- When array is "reasonably short" (fewer than 48 elements) uses Insertion Sort

https://www.toptal.com/developers/sorting-algorithms