Sorting

Data Structures and Algorithms

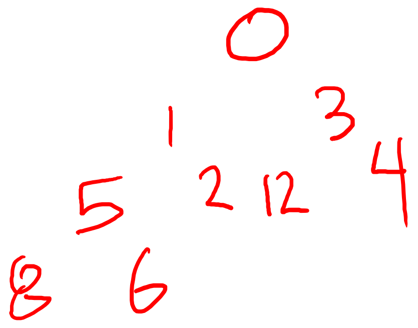
War

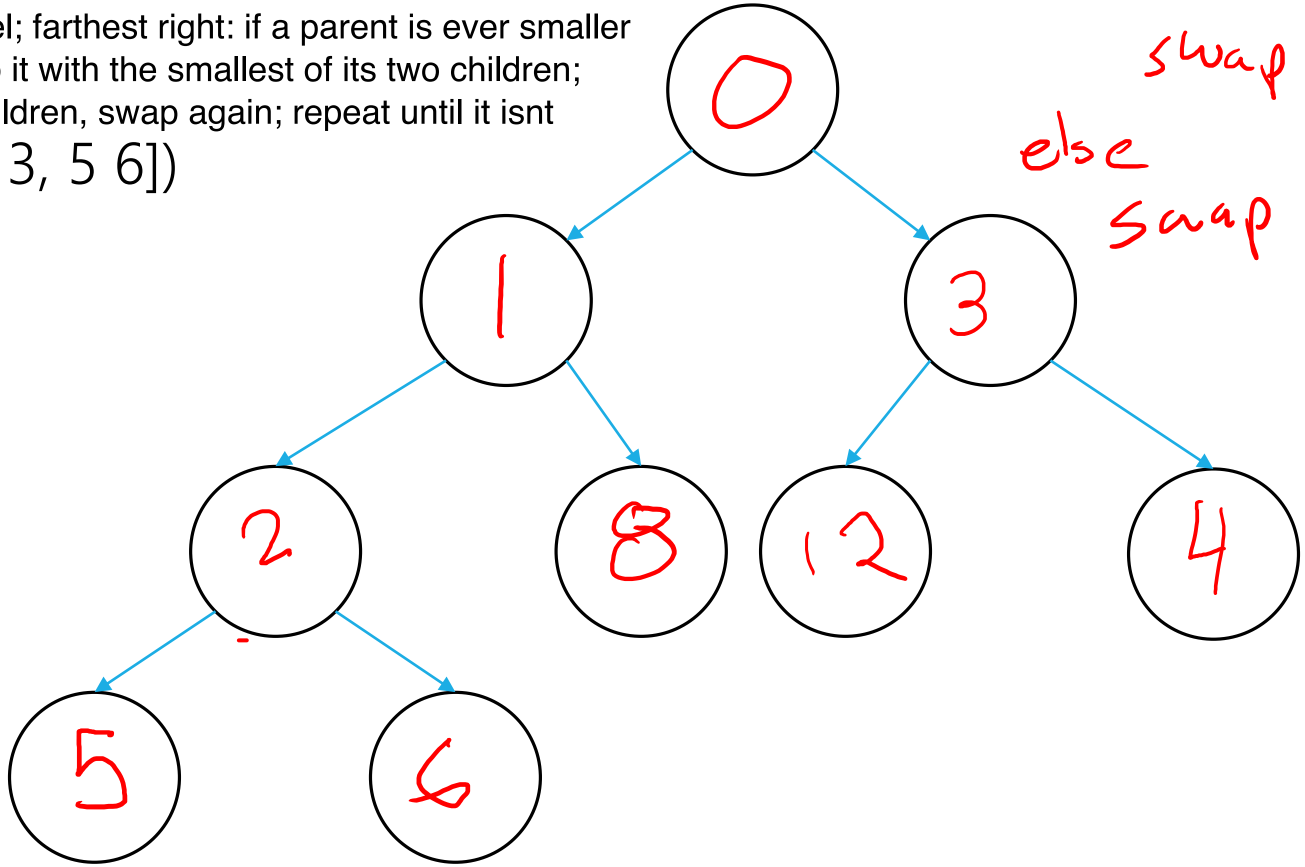
percorlate
DOWN

BUILD HEAP is O(N)
Step 1: Put elements in array (copy over directly)
Step 2: Copy into tree; place elements left to right starting from top then move down one level.
Step 3: Start from lowest level; farthest right: if a parent is ever smaller than one of its children, swap it with the smallest of its two children; if stll larger than one of its children, swap again; repeat until it isnt

buildHeap([8, 2, 4, 0, 1, 12, 3, 5 6])

```
if ( leftchild < rightchild)
    swap left
else
    swap right
```





0	1	3	2	8	12	4	5	6

Sorting

Take this:

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

And make this:

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

Or this:

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

Comparison Sorts

Compare two elements at a time

General sort, works for most types of elements

Element must form a "consistent, total ordering"

For every element a, b and c in the list the following must be true:

- If a <= b and b <= a then a = b
- If a <= b and b <= c then a <= c
- Either a <= b is true or <= a

What does this mean? **compareTo()** works for your elements

Comparison sorts run at fastest **O(nlog(n))** time

Niche Sorts aka "linear sorts"

know extra information about the properties of stuff in your list

Leverages specific properties about the items in the list to achieve faster runtimes

niche sorts typically run O(n) time

In this class we'll focus on comparison sorts

Why Not Just Heap Sort? It’s O(n log n)…

Why not just create more data structure?

In Place sort

more data structures use more memory; why not use data structure elements are already stored in?

A sorting algorithm is in-place if it requires only O(1) extra space to sort the array

Typically modifies the input collection

Useful to minimize memory usage

Stable sort

A sorting algorithm is stable if any equal items remain in the same relative order before and after the sort

Why do we care?

- Sometimes we want to sort based on some, but not all attributes of an item
- Items that “compareTo()” the same might not be exact duplicates
- Enables us to sort on one attribute first then another etc…

[(8, “fox”), (9, “dog”), (4, “wolf”), (8, “cow”)]

[(4, “wolf”), (8, “fox”), (8, “cow”), (9, “dog”)] Stable

[(4, “wolf”), (8, “cow”), (8, “fox”), (9, “dog”)] Unstable

Other Considerations

Worst Case, Average Case

External Sorts (can’t fit in memory)

Good for small data sets

Ease of implementation

LinkedList vs Array

https://www.youtube.com/watch?v=Ns4TPTC8whw

Selection Sort

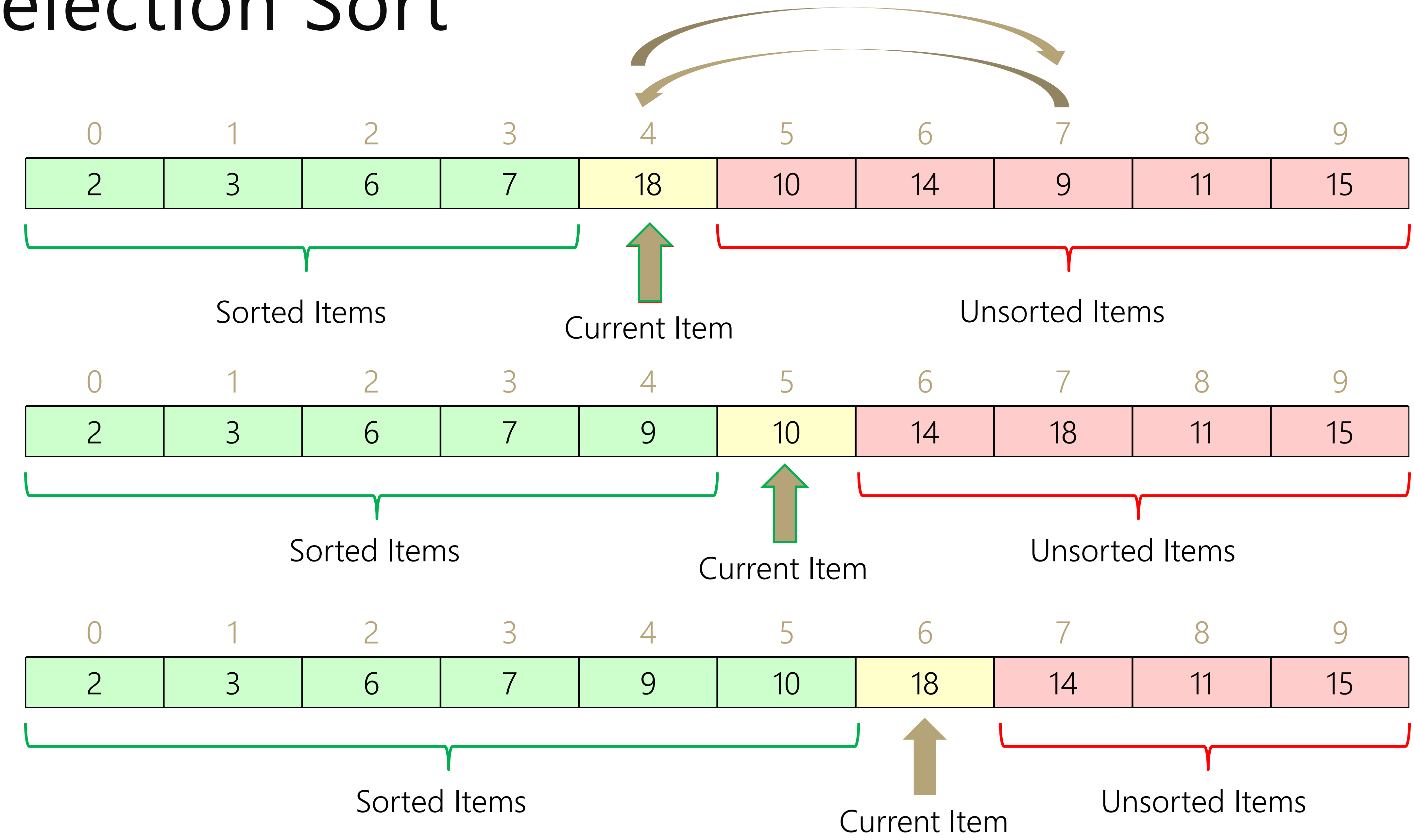
Basic Idea:

Repeatedly scan through the list, moving the next smallest element to the front.

This sounds a lot like heap sort, but worse…

https://www.youtube.com/watch?v=Ns4TPTC8whw

Selection Sort



0	1	2	3	4	5	6	7	8	9
2	3	6	7	18	10	14	9	11	15

Sorted Items

Current Item

Unsorted Items

0	1	2	3	4	5	6	7	8	9
2	3	6	7	9	10	14	18	11	15

Sorted Items

Current Item

Unsorted Items

0	1	2	3	4	5	6	7	8	9
2	3	6	7	9	10	18	14	11	15

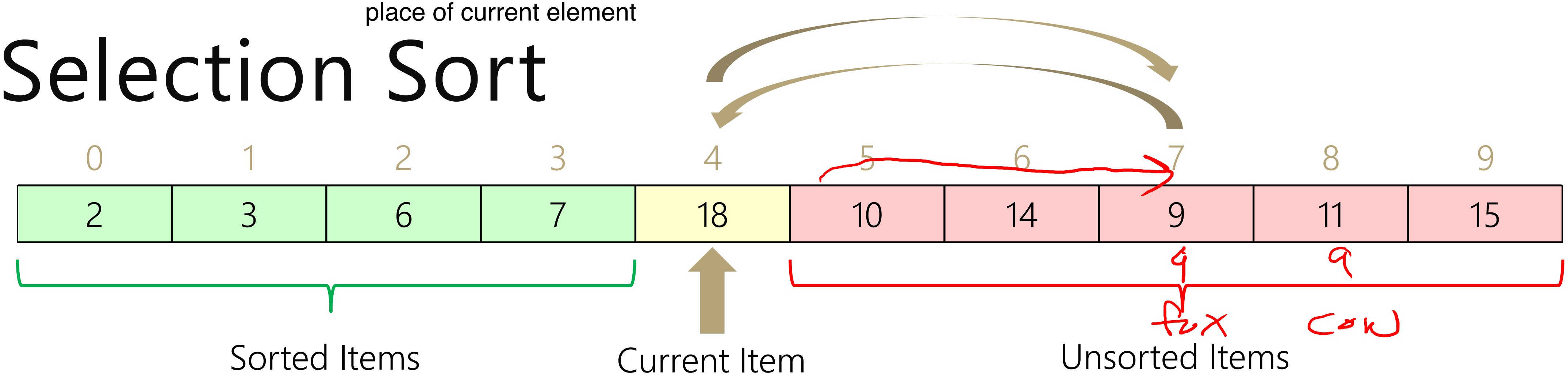
Sorted Items

Current Item

Unsorted Items

Look at current element and all unsorted elements after it; find smallest amongst these, and put it in the place of current element

Selection Sort



0	1	2	3	4	5	6	7	8	9
2	3	6	7	18	10	14	9	11	15

```
public void selectionSort(collection) {
   for (entire list)
      int newIndex = findNextMin(currentItem);
      swap(newIndex, currentItem);
}
public int findNextMin(currentItem) {
   min = currentItem
   for (unsorted list)
      if (item < min)
         min = currentItem
   return min
}
public int swap(newIndex, currentItem) {
   temp = currentItem
   currentItem = newIndex
   newIndex = currentItem
}
```

Always have to traverse over entire unsorted section to find next smallest element

Worst case runtime? $O(n^2)$

Best case runtime? $O(n^2)$

Average runtime? $O(n^2)$

Stable? Yes

In-place? Yes

https://www.youtube.com/watch?v=ROalU379l3U

Insertion Sort

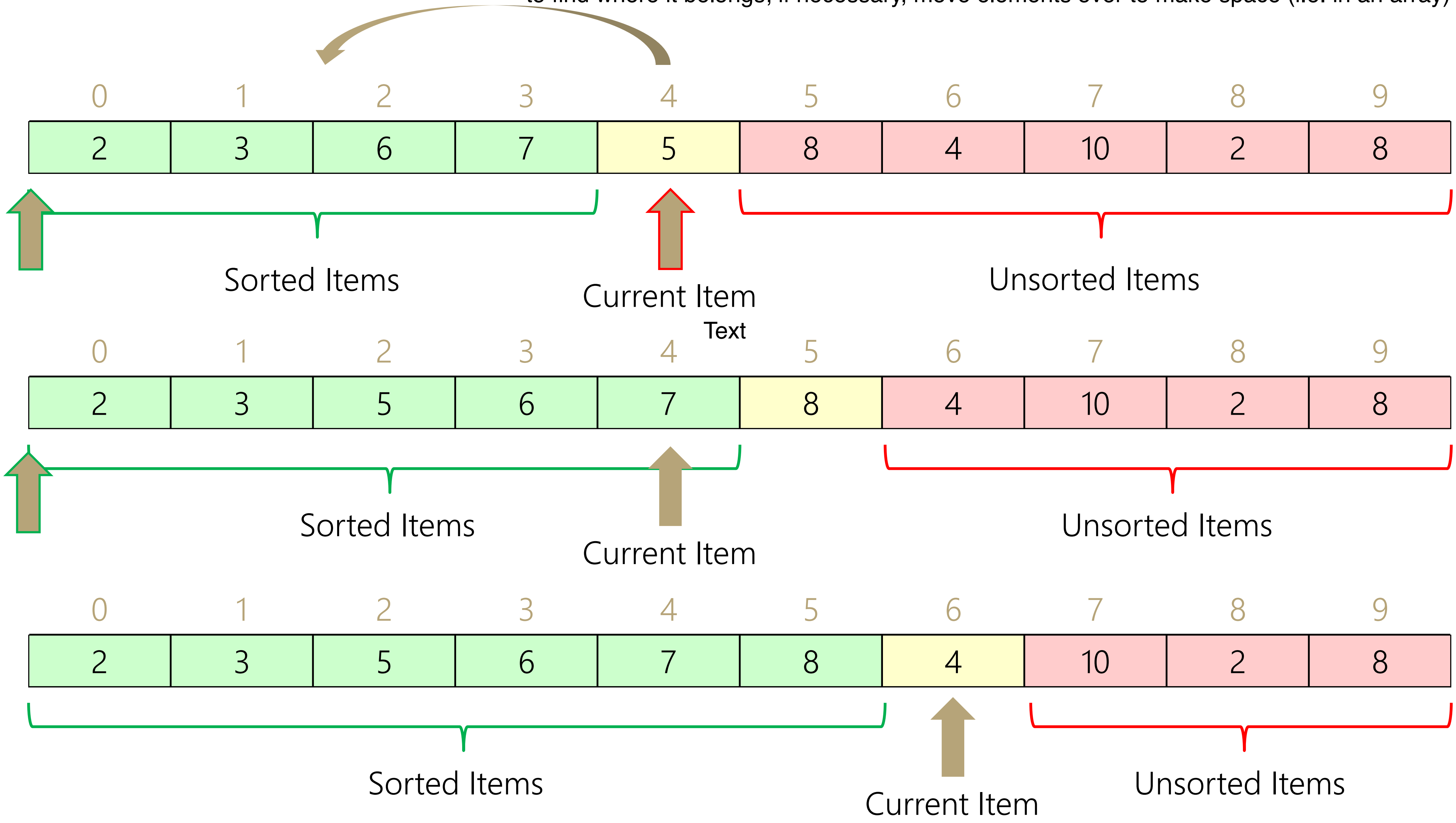
Basic Idea:

Like you would sort a hand of cards – pull out the next card, then insert it into it where it belongs

Insertion Sort

https://www.youtube.com/watch?v=ROalU379l3U

Look at current element, then traverse sorted section (either forward or backward) to find where it belongs; if necessary, move elements over to make space (i.e. in an array)



0	1	2	3	4	5	6	7	8	9
2	3	6	7	5	8	4	10	2	8

Sorted Items

Current Item

Unsorted Items

Text

0	1	2	3	4	5	6	7	8	9
2	3	5	6	7	8	4	10	2	8

Sorted Items

Current Item

Unsorted Items

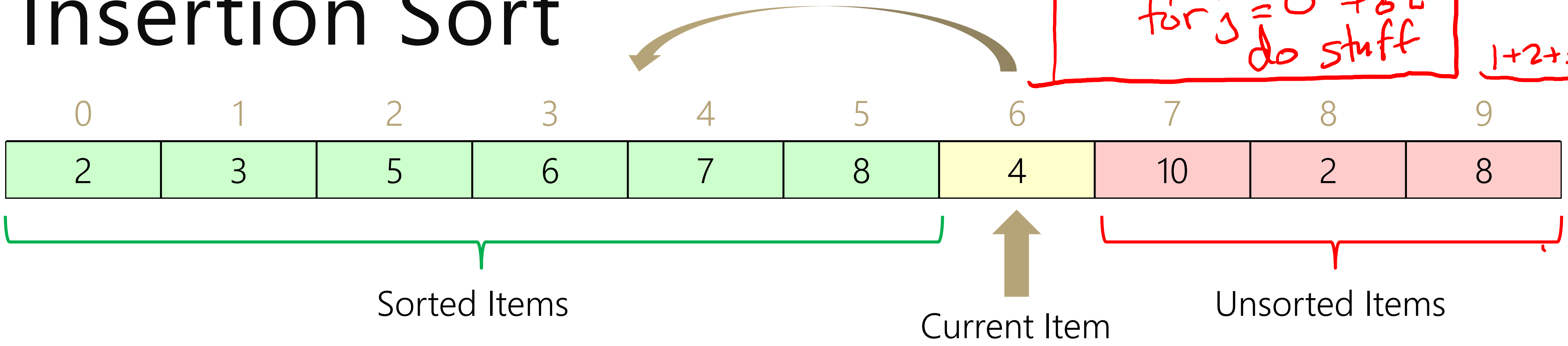
0	1	2	3	4	5	6	7	8	9
2	3	5	6	7	8	4	10	2	8

Sorted Items

Current Item

Unsorted Items

Insertion Sort



0	1	2	3	4	5	6	7	8	9
2	3	5	6	7	8	4	10	2	8

Sorted Items

Current Item

Unsorted Items

```
for i = 1 to n
    for j = 0 to i
        do stuff
```

$$\sum_{i=1}^{n}\sum_{j=0}^{i}$$

$$\underbrace{1+2+3+4+5+\dots+n}_{\frac{(n+1)n}{4} = O(n^2)}$$

1 2 3 4 5 6 7 8 | 6_2

```
public void insertionSort(collection) {
   for (entire list)
      if(currentItem is bigger than nextItem)
         int newIndex = findSpot(currentItem);
         shift(newIndex, currentItem);
}
public int findSpot(currentItem) {
   for (sorted list)
      if (spot found) return
}
public void shift(newIndex, currentItem) {
   for (i = currentItem > newIndex)
      item[i+1] = item[i]
   item[newIndex] = currentItem
}
```

Worst case runtime? $O(n^2)$ if traversing sorted array from left, you have to traverse entire sorted section to find its place

Best case runtime? $O(n)$ traversing sorted array from right; after comparing first elements finds it is in right space

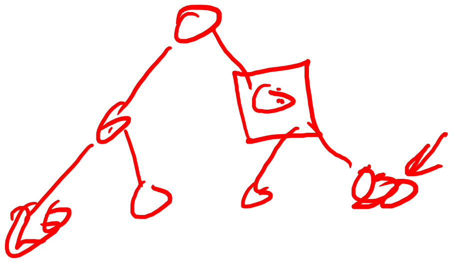
Average runtime? $O(n^2)$

Stable? Yes most element by element sorts are

In-place? Yes

Heap Sort

https://www.youtube.com/watch?v=Xw2D9aJRBY4



1. run Floyd's buildHeap on your data

2. call removeMin n times

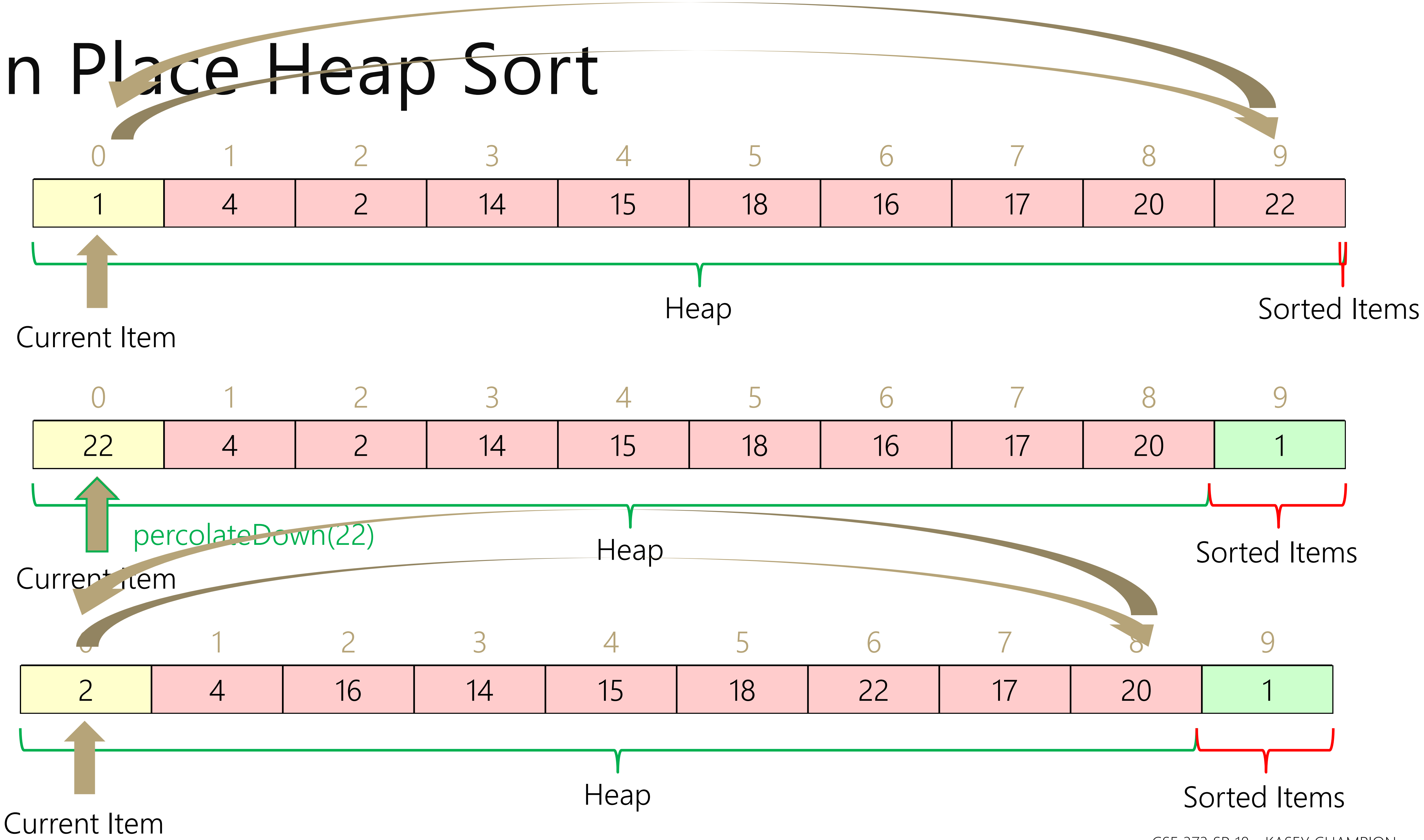
```
public void heapSort(collection) {
   E[] heap = buildHeap(collection)
   E[] output = new E[n]
   for (n)
      output[i] = removeMin(heap)
}
```

creating new data structure; extra memory usage

not much order besides first element; when we percolate, elements to the bottom of tree may move up faster/ slower depending on what is stored above it

Worst case runtime?	O(nlogn)
Best case runtime?	O(nlogn)
Average runtime?	O(nlogn)
Stable?	No
In-place?	No

In Place Heap Sort

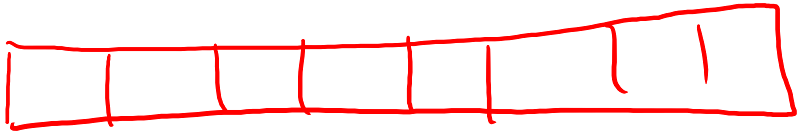


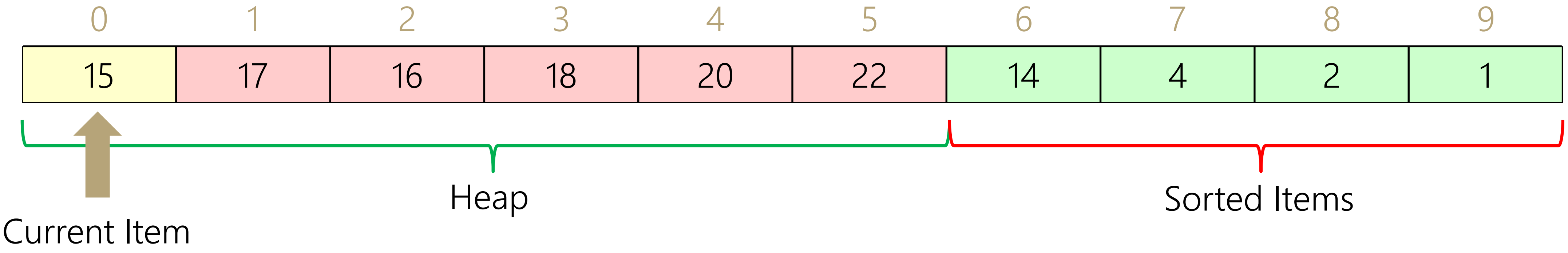
0	1	2	3	4	5	6	7	8	9
1	4	2	14	15	18	16	17	20	22

0	1	2	3	4	5	6	7	8	9
22	4	2	14	15	18	16	17	20	1

0	1	2	3	4	5	6	7	8	9
2	4	16	14	15	18	22	17	20	1

In Place Heap Sort





0	1	2	3	4	5	6	7	8	9
15	17	16	18	20	22	14	4	2	1

```
public void inPlaceHeapSort(collection) {
   E[] heap = buildHeap(collection)
   for (n)
      output[n – i - 1] = removeMin(heap)
}
```

Complication: final array is reversed!

- Run reverse afterwards (O(n))
- Use a max heap
- Reverse compare function to emulate max heap

Worst case runtime?	O(nlogn)
Best case runtime?	O(nlogn)
Average runtime?	O(nlogn)
Stable?	No
In-place?	Yes

Divide and Conquer Technique

$$\underbrace{\boxed{2T(\tfrac{n}{2})}}_{\text{divide!}} + \underbrace{n}_{\text{conquer!}}$$

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

2. Conquer the individual pieces
   - Base case!

3. Combine the results back up recursively

```
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
}
```

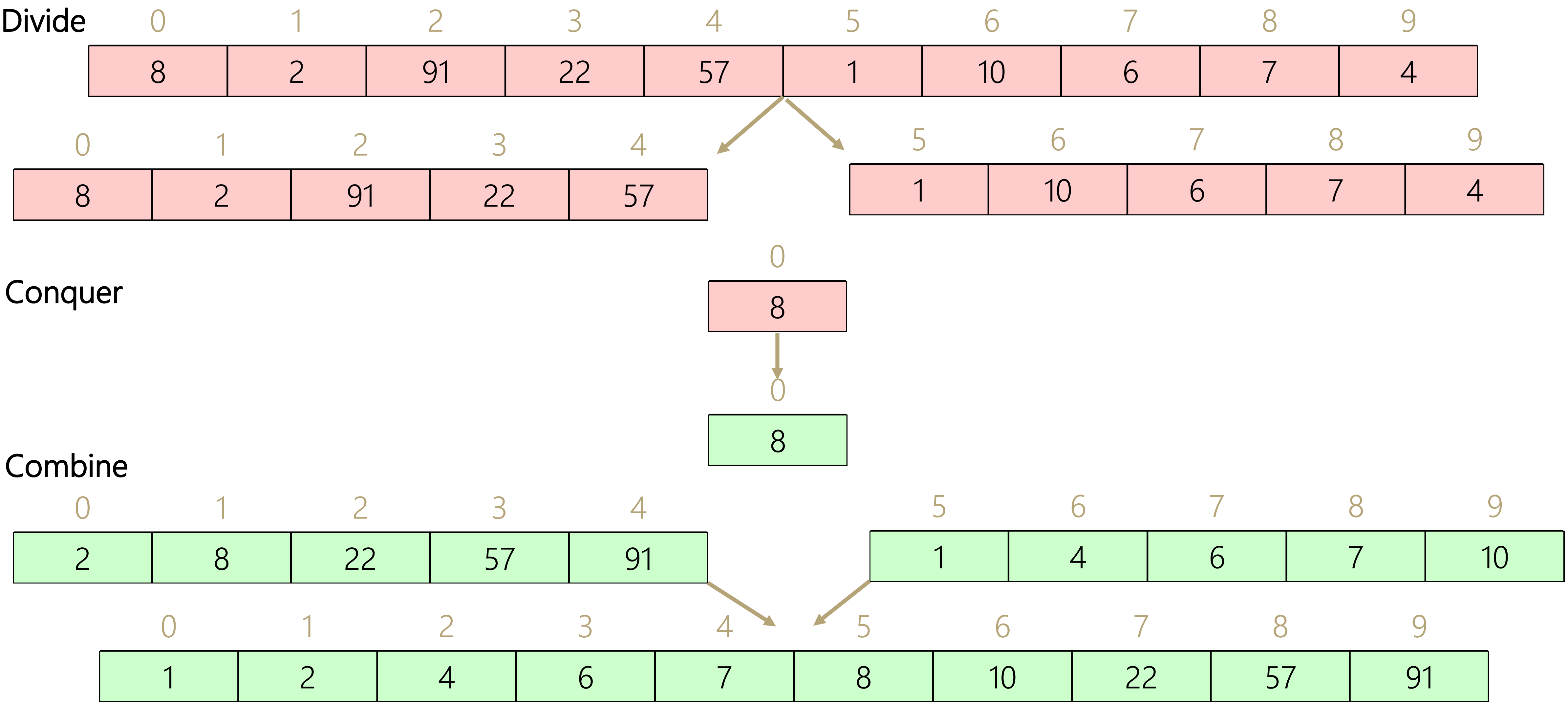
← if size 1 return list

← divide list in pieces, sort recursively

← combine sorted lists

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

Conquer

0
8

0
8

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

$T(1) = 1$

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

$T(1)=1$

$T(n) = 2T\left(\frac{n}{2}\right) \boxed{+n}$

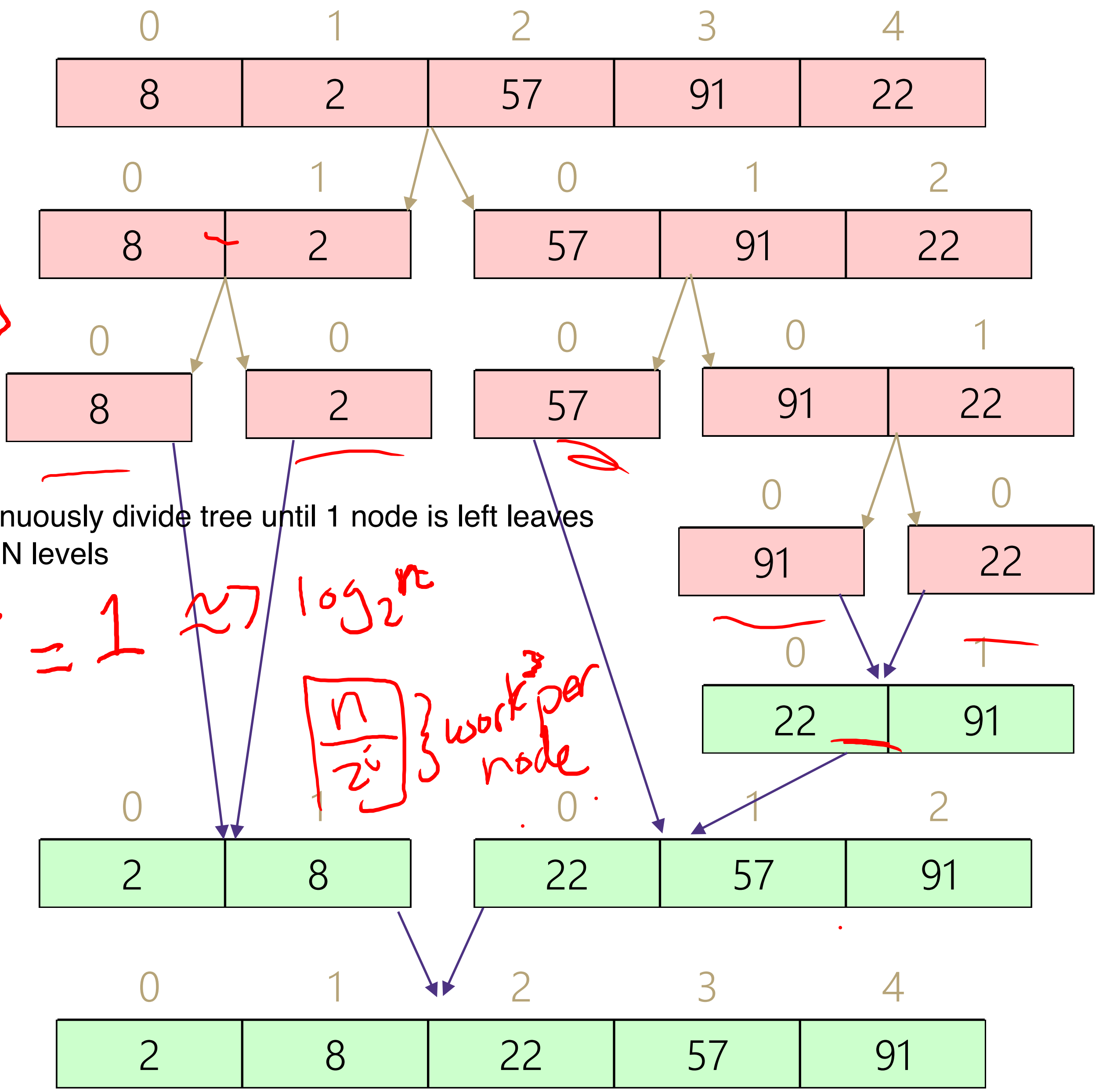
Worst case runtime? $O(n\log n)$

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

Average runtime?

Stable? Yes

In-place? No



continuously divide tree until 1 node is left leaves
log2 N levels

$\frac{n}{2^i} = 1 \Rightarrow \log_2 n$

$\boxed{\frac{n}{2^i}}$ } work per node

At each level you double the number of nodes, and halve the size of each node (i.e. halve the work)

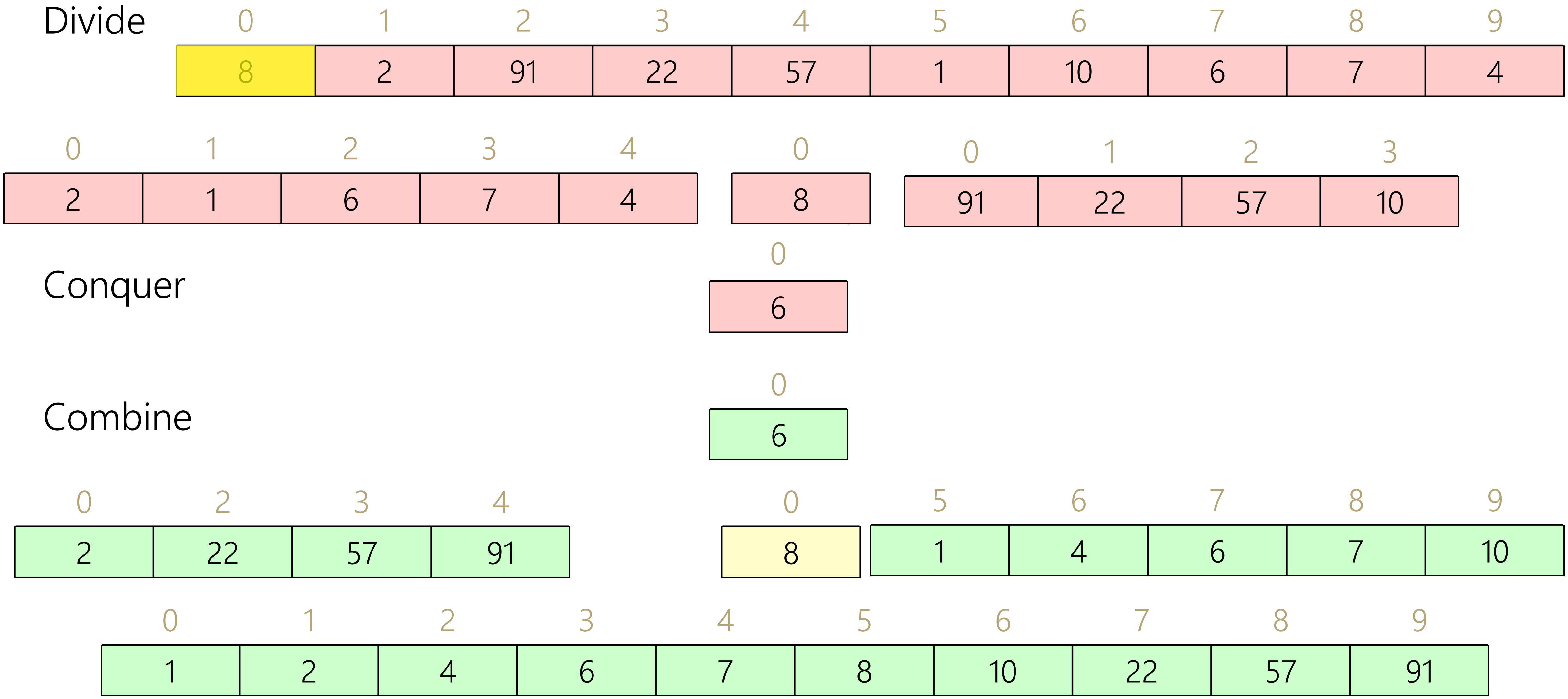
work per layer $= \underbrace{2^i}_{\#\text{ of nodes}} \cdot \underbrace{\frac{n}{2^i}}_{\text{work/node}} = n$

Merge Sort Optimization

Use just two arrays – swap between them

Quick Sort

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

Conquer

0
6

Combine

0
6

0	2	3	4
2	22	57	91

0
8

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

Quick Sort

```
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? $T(n) = \begin{cases} 1 \text{ if } n <= 1 \\ n + T(n-1) \text{ otherwise} \end{cases}$

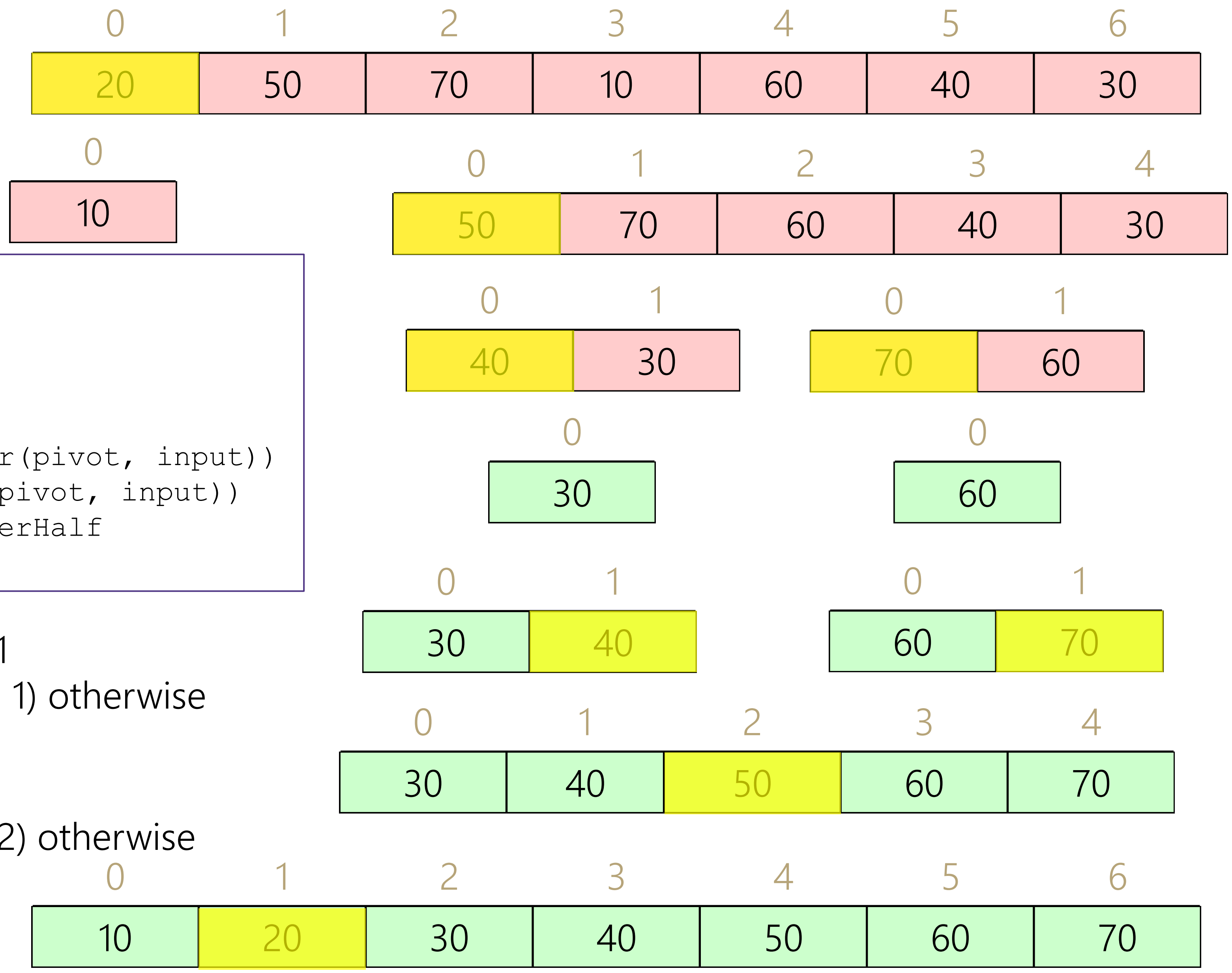
Best case runtime?

Average runtime?

$T(n) = \begin{cases} 1 \text{ if } n <= 1 \\ n + 2T(n/2) \text{ otherwise} \end{cases}$

Stable? No

In-place? No



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

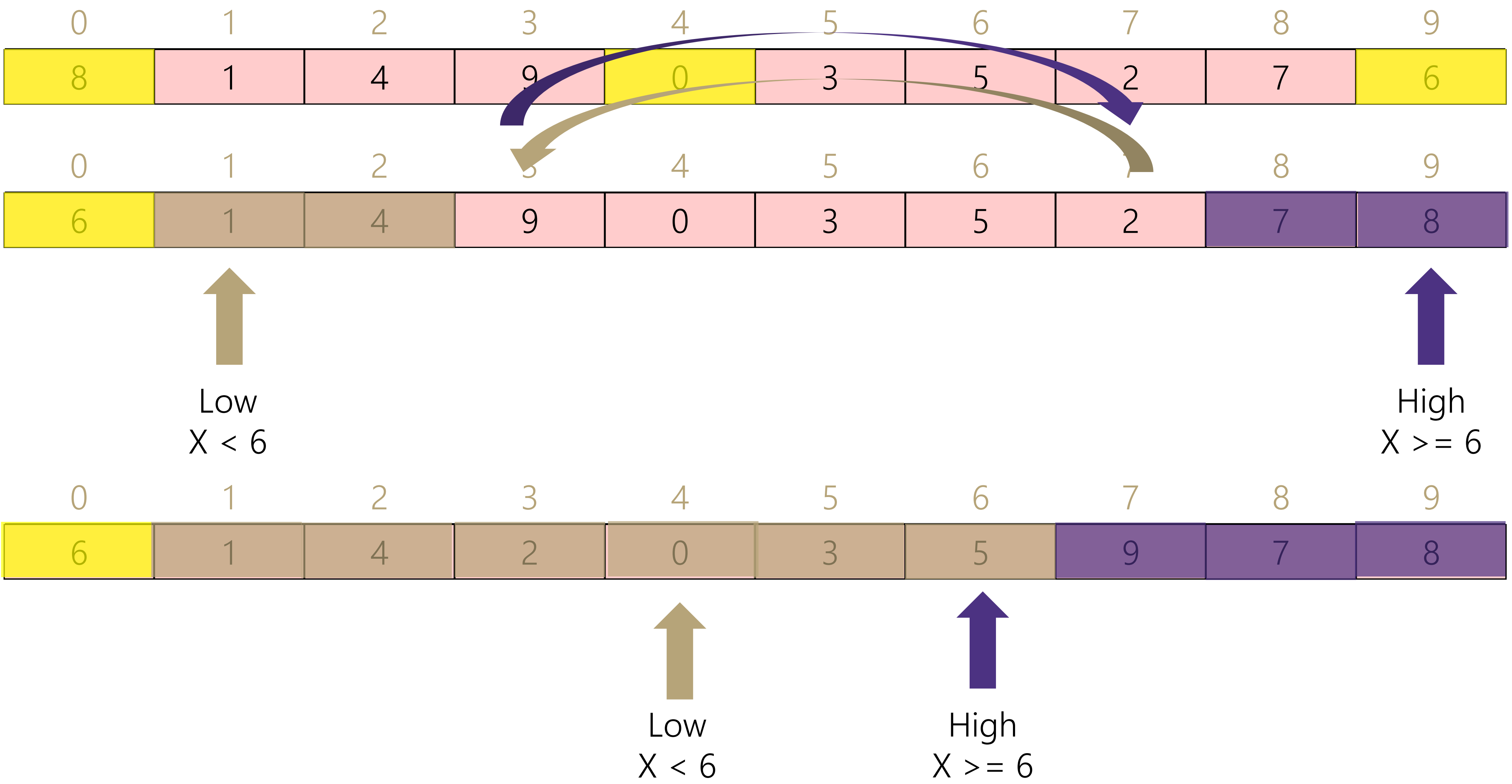
Can we do better?

Pick a better pivot

- Pick a random number
- Pick the median of the first, middle and last element

Sort elements by swapping around pivot in place

Better Quick Sort



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

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

Announcements

Project 1 (Calculator) is Due Tonight! Use “SUBMIT” as tag.

HW3 (Individual Assignment) will be assigned this weekend

- Due Sunday 7/22
- Make sure you know how to do it before the midterm! It’s the best midterm review

Come to Class Next Week:

- Monday: Midterm Review – Going from Diagrams to Code
- Wednesday: Software Engineering – Deep Dive into Git, Pair Programming, and Testing
- Friday: Midterm Exam! – Review materials and practice midterms on website (this evening).

More Announcements

If you are applying to the CSE major, send me an e-mail reminding me of our interactions

Office hours immediately after class – follow me!