# Lecture 22: Introduction to Sorting

CSE 373: Data Structures and Algorithms

# Administrivia

Assignment Reminders

- Project 4 due Wednesday June 2nd
- Exercise 4 due Friday May 21st

- Grades coming this week

# Sorting


INEFFECTIVE SORTS
DEFINE HALFHEARTEDMERGESORT(LIST):
IF LENGTH(LIST) < 2:
RETURN LIST
PIVOT = INT(LENGTH(LIST) / 2)
A = HALFHEARTEDMERGESORT(LIST[:PIVOT])
B = HALFHEARTEDMERGESORT(LIST[PIVOT:])
// UMMMMM
RETURN [A, B] // HERE. SORRY.
DEFINE FASTBOGOSORT(LIST):
// AN OPTIMIZED BOGOSORT
// RUNS IN O(N LOGN)
FOR N FROM 1 TO LOG(LENGTH(LIST)):
SHUFFLE(LIST):
IF ISSORTED(LIST):
RETURN LIST
RETURN "KERNEL PAGE FAULT (ERROR CODE: 2)"
DEFINE JOBINTERVIEWQUICKSORT(LIST):
OK SO YOU CHOOSE A PIVOT
THEN DIVIDE THE LIST IN HALF
FOR EACH HALF:
CHECK TO SEE IF IT'S SORTED
NO, WAIT, IT DOESN'T MATTER
COMPARE EACH ELEMENT TO THE PIVOT
THE BIGGER ONES GO IN A NEW LIST
THE EQUAL ONES GO INTO, UH
THE SECOND LIST FROM BEFORE
HANG ON, LET ME NAME THE LISTS
THIS IS LIST A
THE NEW ONE IS LIST B
PUT THE BIG ONES INTO LIST B
NOW TAKE THE SECOND LIST
CALL IT LIST, UH, A2
WHICH ONE WAS THE PIVOT IN?
SCRATCH ALL THAT
IT JUST RECURSIVELY CALLS ITSELF
UNTIL BOTH LISTS ARE EMPTY
RIGHT?
NOT EMPTY, BUT YOU KNOW WHAT I MEAN
AM I ALLOWED TO USE THE STANDARD LIBRARIES?
DEFINE PANICSORT(LIST):
IF ISSORTED(LIST):
RETURN LIST
FOR N FROM 1 TO 10000:
PIVOT = RANDOM(0, LENGTH(LIST))
LIST = LIST[PIVOT:] + LIST[:PIVOT]
IF ISSORTED(LIST):
RETURN LIST
IF ISSORTED(LIST):
RETURN LIST:
IF ISSORTED(LIST): //THIS CAN'T BE HAPPENING
RETURN LIST
IF ISSORTED(LIST): // COME ON COME ON
RETURN LIST
// OH JEEZ
// I'M GONNA BE IN SO MUCH TROUBLE
LIST = [ ]
SYSTEM("SHUTDOWN -H +5")
SYSTEM("RM -RF ./")
SYSTEM("RM -RF ~/*")
SYSTEM("RM -RF /")
SYSTEM("RD /S /Q C:*") //PORTABILITY
RETURN [1, 2, 3, 4, 5]

# Where are we?

This course is "data structures and algorithms"

Data structures

- Organize our data so we can process it effectively

Algorithms

- Actually process our data!

We're going to start focusing on algorithms

We'll start with sorting

- A very common, generally-useful preprocessing step
- And a convenient way to discuss a few different ideas for designing algorithms.

# Types of Sorts

## Comparison Sorts

Compare two elements at a time

General sort, works for most types of elements

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

- And for our running times to be correct, compareTo must run in $O(1)$ time.

## Niche Sorts aka "linear sorts"

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

niche sorts typically run O(n) time

For example, we're sorting small integers, or short strings.

In this class we'll focus on comparison sorts

# Sorting: Definitions (Knuth's TAOCP)

An **ordering relation** < for keys a, b, and c has the following properties:

- Law of Trichotomy: Exactly one of a < b, a = b, b < a is true
- Law of Transitivity: If a < b, and b < c, then a < c

A **sort** is a permutation (re-arrangement) of a sequence of elements that puts the keys into non-decreasing order, relative to the ordering relation

- $x_1 \leq x_2 \leq x_3 \leq \ldots \leq x_N$

```
int temperature
```

- Built-in, simple ordering relation

```
class Movie {
  String name;
  int year;
}
```

- More complex: Whenever we sort, *we also must decide* what ordering relation to use for that application
  - Sort by name?
  - Sort by year?
  - Some combination of both?

# Sorting: Stability

A sort is stable if the relative order of *equivalent* keys is maintained after sorting

INPUT

| | | | | | |
|---|---|---|---|---|---|
| Anita *2010* | Basia *2018* | Caris *2019* | Duska *2020* | Duska *2015* | Anita *2016* |

**Stable** sort using name as key

| | | | | | |
|---|---|---|---|---|---|
| Anita *2010* | Anita *2016* | Basia *2018* | Caris *2019* | Duska *2020* | Duska *2015* |

**Unstable** sort using name as key

| | | | | | |
|---|---|---|---|---|---|
| Anita *2016* | Anita *2010* | Basia *2018* | Caris *2019* | Duska *2015* | Duska *2020* |

- Stability and Equivalency only matter for complex types
  - i.e. when there is more data than just the key

| | | | | | | |
|---|---|---|---|---|---|---|
| Anita | Basia | Anita | Duska | Esteban | Duska | Caris |

| | | | | | | |
|---|---|---|---|---|---|---|
| Anita | Anita | Basia | Caris | Duska | Duska | Esteban |

# Sorting: Performance Definitions

Runtime performance is sometimes called the **time complexity**

- Example: Dijkstra's has time complexity O(E log V).

<u>Extra</u> memory usage is sometimes called the **space complexity**

- Dijkstra's has space complexity Θ(V)
  - Priority Queue, distTo and edgeTo maps
- The input graph takes up space Θ(V+E), but we don't count this as part of the space complexity since the graph itself already exists and is an input to Dijkstra's

# Sorting Goals

## In Place sort

A sorting algorithm is in-place if it allocates $O(1)$ extra memory

Modifies input array (can't copy data into new array)

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?

- "data exploration" Client code will want to sort by multiple features and "break ties" with secondary features

[**(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

## Speed

Of course, we want our algorithms to be fast.

Sorting is so common, that we often start caring about constant factors.

# SO MANY SORTS

Quicksort, Merge sort, in-place merge sort, heap sort, insertion sort, intro sort, selection sort, timsort, cubesort, shell sort, bubble sort, binary tree sort, cycle sort, library sort, patience sorting, smoothsort, strand sort, tournament sort, cocktail sort, comb sort, gnome sort, block sort, stackoverflow sort, odd-even sort, pigeonhole sort, bucket sort, counting sort, radix sort, spreadsort, burstsort, flashsort, postman sort, bead sort, simple pancake sort, spaghetti sort, sorting network, bitonic sort, bogosort, stooge sort, insertion sort, slow sort, rainbow sort…

# Goals

Algorithm Design (like writing invariants) is more art than science.

We'll do a little bit of designing our own algorithms

- Take CSE 417 (usually runs in Winter) for more

Mostly we'll understand how existing algorithms work

Understand their pros and cons

- Design decisions!

Practice how to apply those algorithms to solve problems

# Algorithm Design Patterns

Algorithms don't just come out of thin air.

There are common patterns we use to design new algorithms.

Many of them are applicable to sorting (we'll see more patterns later in the quarter)

Invariants/Iterative improvement

- Step-by-step make one more part of the input your desired output.

Using data structures

- Speed up our existing ideas

Divide and conquer

- Split your input
- Solve each part (recursively)
- Combine solved parts into a single

# Principle 1

Invariants/Iterative improvement

- Step-by-step make one more part of the input your desired output.

We'll write iterative algorithms to satisfy the following invariant:

After $k$ iterations of the loop, the first $k$ elements of the array will be sorted.

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 | 14 | 18 | 11 | 15 |

Sorted Items

Current Item

Unsorted Items

# 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

```
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
}
```

Worst case runtime? $\Theta(n^2)$

Best case runtime? $\Theta(n^2)$

In-practice runtime? $\Theta(n^2)$

Stable? No

In-place? Yes

# Selection Sort Stability

| 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| 5a | 3 | 4 | 5b | 2 | 6 | 8 |

√

| 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| 2 | 3 | 4 | 5b | 5a | 6 | 8 |

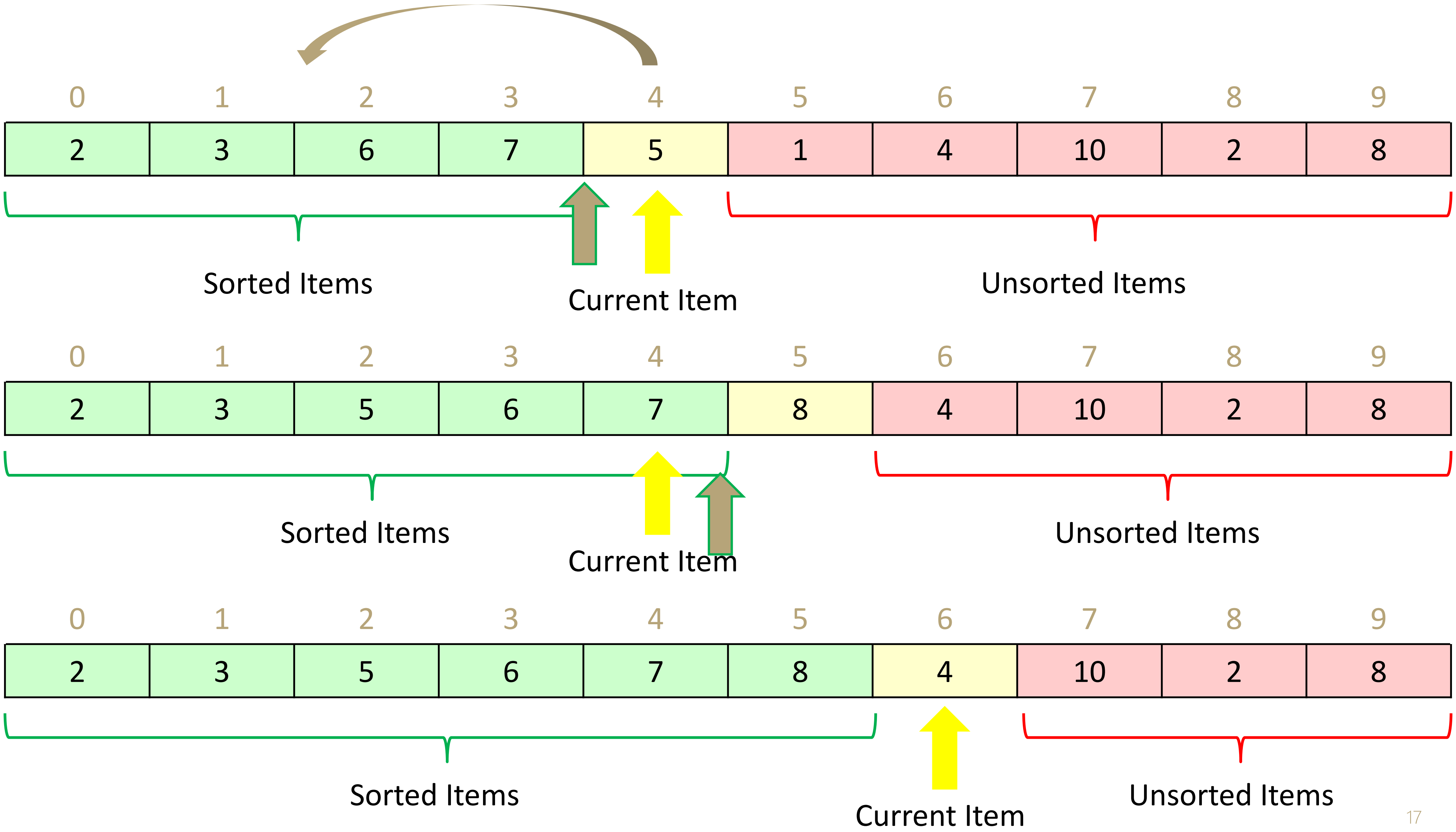
…

| 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| 2 | 3 | 4 | 5b | 5a | 6 | 8 |

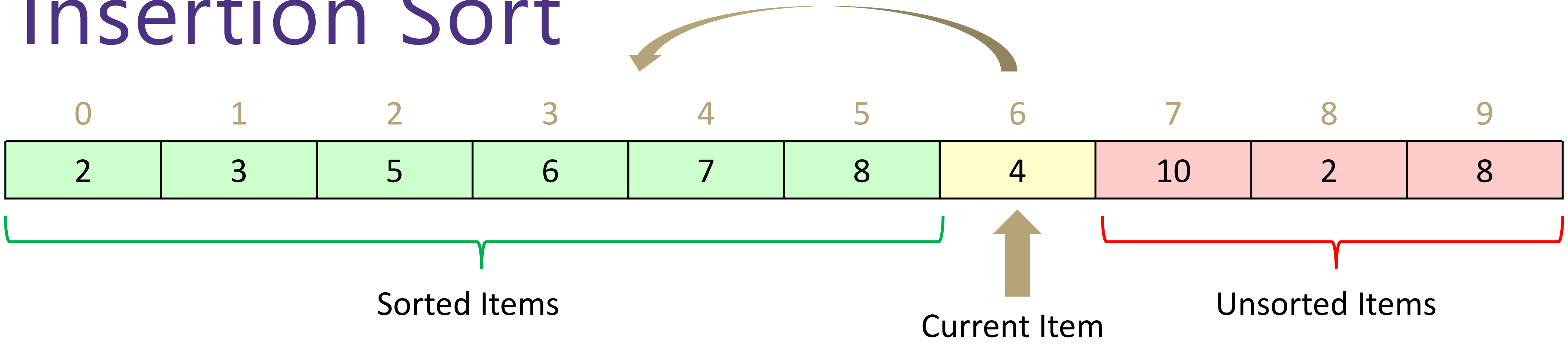
*Swapping non-adjacent items can result in instability of sorting algorithms

# Insertion Sort

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



# Insertion Sort




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

Worst case runtime? $\Theta(n^2)$

Best case runtime? $\Theta(n)$

In-practice runtime? $\Theta(n^2)$

Stable? Yes

In-place? Yes

# Insertion Sort Stability

| 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| 5a | 3 | 4 | 5b | 2 | 6 | 8 |

| 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| 5a | 3 | 4 | 5b | 2 | 6 | 8 |

| 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| 3 | 5a | 4 | 5b | 2 | 6 | 8 |

| 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| 3 | 4 | 5a | 5b | 2 | 6 | 8 |

| 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| 3 | 4 | 5a | 5b | 2 | 6 | 8 |

Insertion sort is stable

- All swaps happen between adjacent items to get current item into correct relative position within sorted portion of array
- Duplicates will always be compared against one another in their original orientation, thus it can be maintained with proper if logic

# Principle 2

Selection sort:

After $k$ iterations of the loop, the $k$ smallest elements of the array are (sorted) in indices $0, \ldots, k-1$

Runs in $\Theta(n^2)$ time no matter what.

Using data structures

- Speed up our existing ideas

If only we had a data structure that was good at getting the smallest item remaining in our dataset…

- We do!

# 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(input) {
   E[] heap = buildHeap(input)
   E[] output = new E[n]
   for (n)
      output[i] = removeMin(heap)
}
```

| | |
|---|---|
| Worst case runtime? | $\Theta(n \log n)$ |
| Best case runtime? | $\Theta(n)$ |
| In-practice runtime? | $\Theta(n \log n)$ |
| Stable? | No |
| In-place? | If we get clever... |

# In Place Heap Sort

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

Current Item

Heap

Sorted Items

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

percolateDown(22)

Current Item

Heap

Sorted Items

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

Current Item

Heap

Sorted Items

# In Place Heap Sort

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

Current Item

Heap

Sorted Items

```
public void inPlaceHeapSort(input) {
   buildHeap(input) // alters original array
   for (n : input)
      input[n – i - 1] = removeMin(heap)
}
```

Complication: final array is reversed! Lots of fixes:

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

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

Best case runtime? $\Theta(n)$

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

Stable? No

In-place? Yes