



Middlebury

CSCI 201: Data Structures

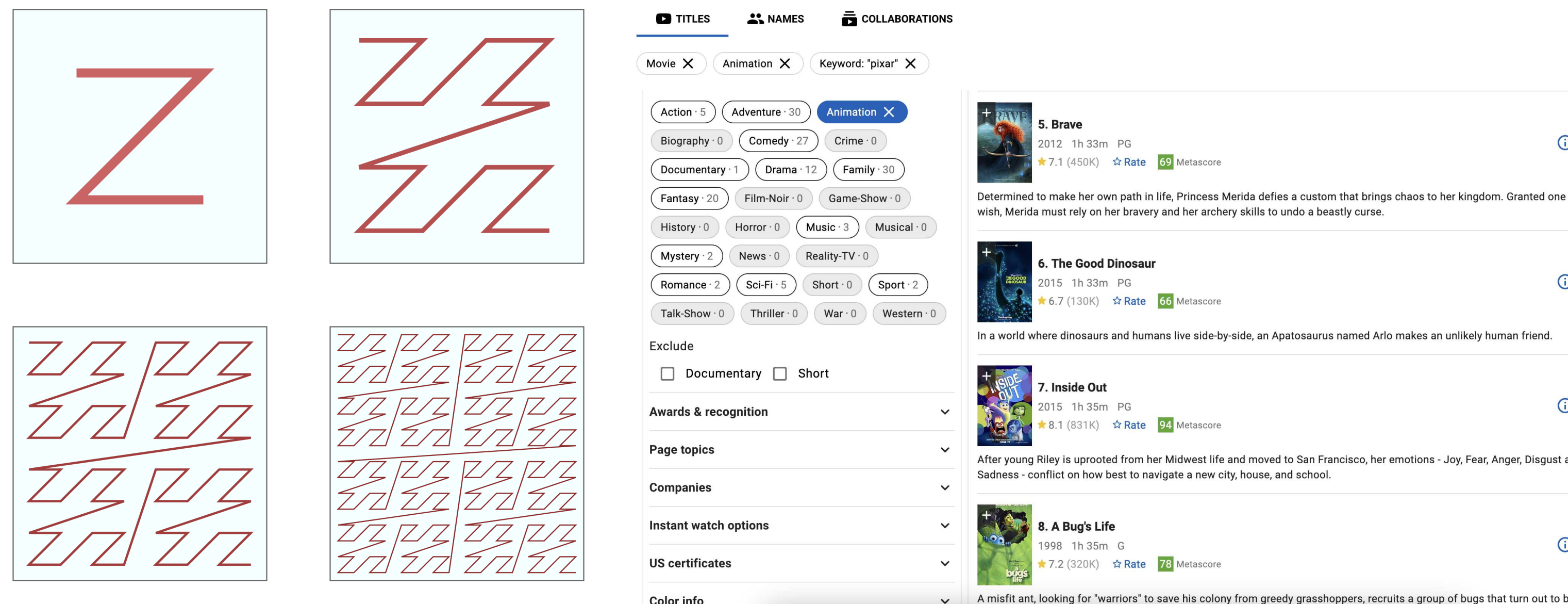
Fall 2024

Lecture 4R: Sorting

Goals for today:

- Analyze the runtime of sorting algorithms including **selection sort** and **insertion sort**.
- Describe the steps in **bucket sort** and **radix sort**.
- Differentiate between the **best**, **worst** and **average** case runtime of an algorithm.
- Identify properties of sorting algorithms: **in-place**, **stable**.
- Customize how sorting is done for our own objects.
- Pair program to implement a sorting algorithm! 🐸

Why is sorting important?



The image illustrates the importance of sorting through two visual examples. On the left, four square panels show the progression of a sorting algorithm: 1. A single red 'Z' shape. 2. Two red 'Z' shapes. 3. Four red 'Z' shapes. 4. Eight red 'Z' shapes arranged in a grid. On the right, a screenshot of the IMDb website shows a list of animated movies, with 'Brave' at the top, followed by 'The Good Dinosaur', 'Inside Out', and 'A Bug's Life'.

Refresher exercise from last class: determine $T(n)$ (an expression for the number of operations performed by the following algorithm), then provide a big-oh bound on $T(n)$.

eg. $n = 32$

count # divisions
generalize to any n

```
int i = n;  
while (i > 1) {  
    i = i / 2;  
}
```

let $n = 32$

$i = n = 32$	$i > 1?$ yes
$i = i/2 = 16$	$i > 1?$ yes
$i = i/2 = 8$	$i > 1?$ yes
$i = i/2 = 4$	$i > 1?$ yes
$i = i/2 = 2$	$i > 1?$ yes
$i = i/2 = 1$	$i > 1?$ no

break while loop

relationship between #divisions (d)
and n ?

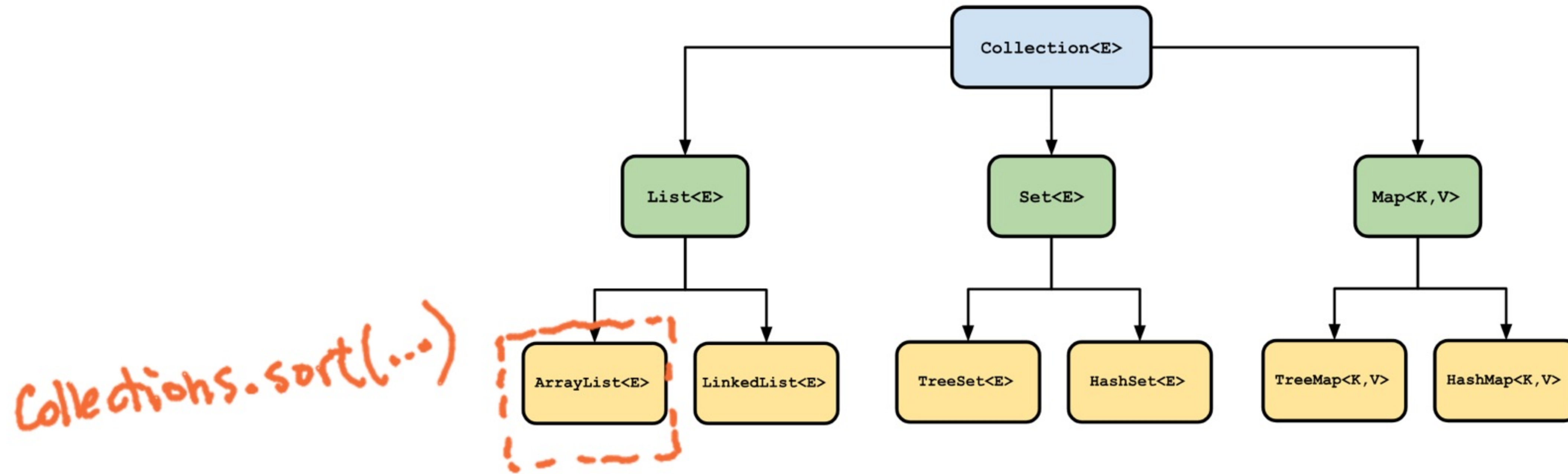
$$2^d = n$$

$$d = \log_2(n) = T(n)$$

$O(\log n)$



The **Collections** framework has built-in methods to **sort**.



`public static void sort(List<T> list)`

Sorts the specified list into ascending order, according to the natural ordering of its elements. All elements in the list must implement the **Comparable** interface.

This sort is guaranteed to be stable: equal elements will not be reordered as a result of the sort.

4, 2, 2, 1
1, 2, 2, 4 < >

The **Arrays** class also has built-in **static** methods to **sort** which can be used for fixed-size arrays.

static void	sort (double[] a) Sorts the specified array into ascending numerical order.
static void	sort (double[] a, int fromIndex, int toIndex) Sorts the specified range of the array into ascending order.
static void	sort (float[] a) Sorts the specified array into ascending numerical order.
static void	sort (float[] a, int fromIndex, int toIndex) Sorts the specified range of the array into ascending order.
static void	sort (int[] a) Sorts the specified array into ascending numerical order.
static void	sort (int[] a, int fromIndex, int toIndex) Sorts the specified range of the array into ascending order.
static void	sort (long[] a) Sorts the specified array into ascending numerical order.
static void	sort (long[] a, int fromIndex, int toIndex) Sorts the specified range of the array into ascending order.
static void	sort (Object[] a) Sorts the specified array of objects into ascending order, according to the natural ordering of its elements.
static void	sort (Object[] a, int fromIndex, int toIndex) Sorts the specified range of the specified array of objects into ascending order, according to the natural ordering of its elements.
static void	sort (short[] a) Sorts the specified array into ascending numerical order.
static void	sort (short[] a, int fromIndex, int toIndex) Sorts the specified range of the array into ascending order.
static <T> void	sort (T[] a, Comparator <? super T> c) Sorts the specified array of objects according to the order induced by the specified comparator.
static <T> void	sort (T[] a, int fromIndex, int toIndex, Comparator <? super T> c) Sorts the specified range of the specified array of objects according to the order induced by the specified comparator.

Arrays.sort(...)

we'll also talk
comparators
today.

<https://docs.oracle.com/javase/8/docs/api/java/util/Arrays.html>

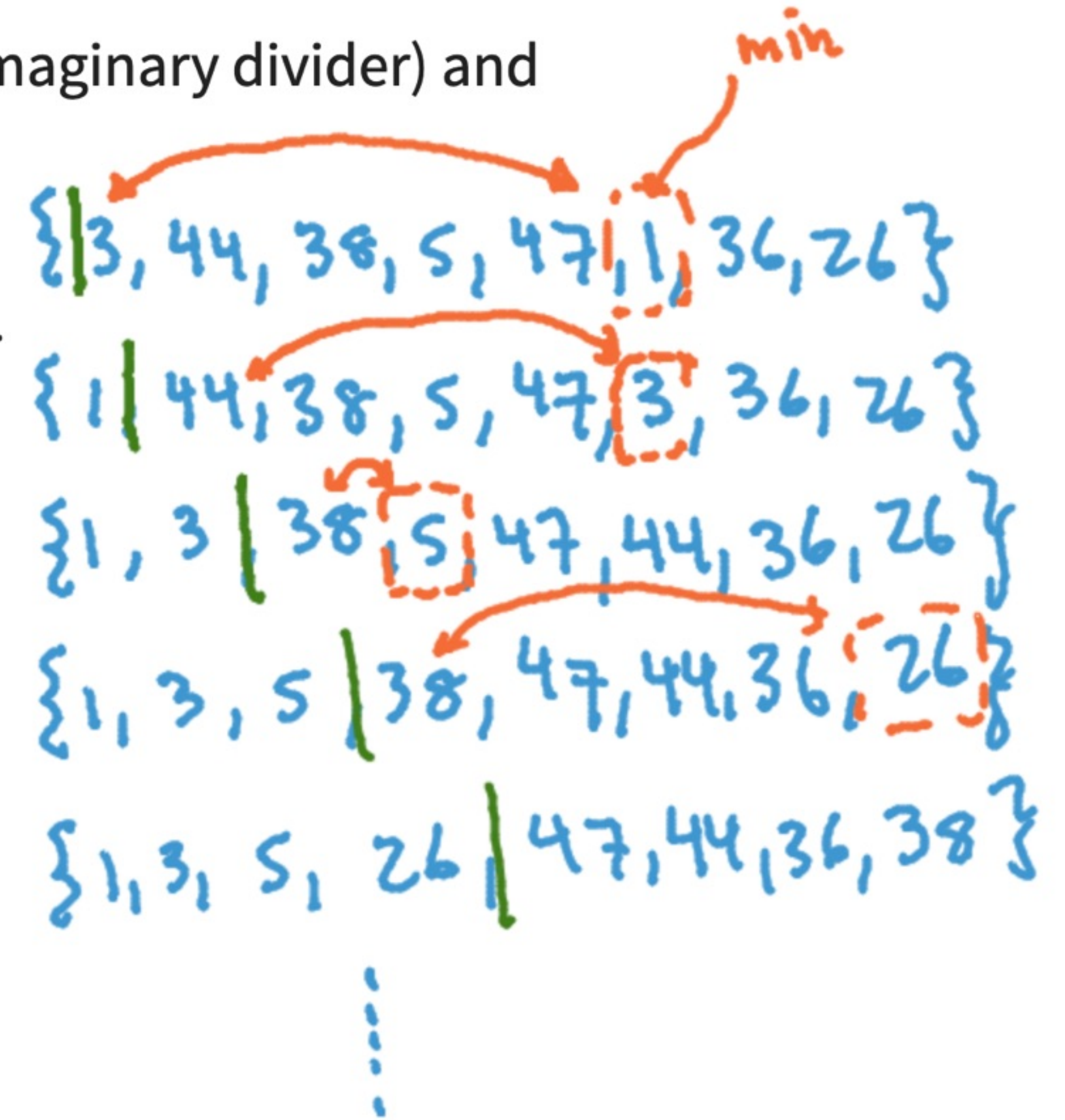


Sorting Algorithm #1 (Selection Sort):

Main idea: maintain sorted elements on the left (of some imaginary divider) and unsorted elements on the right.

1. Find the smallest element in unsorted part.
2. Swap this smallest element with the element to the right of this divider.
3. Move the divider to the right (by one) and go back to Step 1.

```
1 public static void sort(int[] items) {  
2     for (int i = 0; i < items.length; i++) {  
3         int minValue = items[i];  
4         int minIndex = i;  
5         for (int j = i + 1; j < items.length; j++) {  
6             if (items[j] < minValue) {  
7                 minValue = items[j];  
8                 minIndex = j;  
9             }  
10        }  
11        items[minIndex] = items[i];  
12        items[i] = minValue;  
13    }  
14 }
```



Sorting Algorithm #2 (Insertion Sort):

Main idea: maintain sorted elements on the left (of some imaginary divider) and unsorted elements on the right.

1. Look at first element in unsorted part (to the right of divider).
2. Iteratively swap this into the correct place in the sorted part.
3. Move the divider to the right (by one) and go back to Step 1.

{ 3, 44, 38, 5, 47, 1, 36, 26 }

{ 3, 44 | 38, 5, 47, 1, 36, 26 }


{ 3, 38, 44 | 5, 47, 1, 36, 26 }

{ 3, 5, 38, 44 | 47, 1, 36, 26 }

⋮

Work in pairs in **InsertionSort.java!** 

< 0 0 0 0 Live Share Java: Ready

 Invitation link copied to clipboard! Send it to anyone you trust or click "More info" to learn about secure sharing.

Source: Live Share

Make read-only

More info

Copy again

1 person
clicks
this,

then
send

Possible implementation of **InsertionSort.java**.

```
public static void sort(int[] items) {  
    for (int i = 0; i < items.length; i++) {  
        int j = i;  
        while (j > 0 && items[j] < items[j - 1]) {  
            // swap items at j and j - 1  
            int tmp = items[j];  
            items[j] = items[j - 1];  
            items[j - 1] = tmp;  
            j--;  
        }  
    }  
}
```

] other implementations
possible that
make this more efficient.

Runtime analysis of selection and insertion sort.

```
// selection sort
public static void sort(int[] items) {
    for (int i = 0; i < items.length; i++) {
        int minValue = items[i];
        int minIndex = i;
        for (int j = i + 1; j < items.length; j++) {
            if (items[j] < minValue) {
                minValue = items[j];
                minIndex = j;
            }
        }
        items[minIndex] = items[i];
        items[i] = minValue;
    }
}
```

```
// insertion sort
public static void sort(int[] items) {
    for (int i = 0; i < items.length; i++) {
        int j = i;
        while (j > 0 && items[j] < items[j - 1]) {
            // swap items at j and j - 1
            int tmp = items[j];
            items[j] = items[j - 1];
            items[j - 1] = tmp;
            j--;
        }
    }
}
```

comparisons

i = 0
+ n - 1
i = 1
+ n - 2
i = 2
+ n - 3
⋮
i = n - 2
+ 1
i = n - 1
+ 0

looks
arithmetic-y
 $\approx \frac{n(n+1)}{2}$
 $O(n^2)$ average worst

comparisons

i = 0
+ 0
i = 1
+ 1
i = 2
+ 2
⋮
+ n - 1
i = n - 1

arithmetic-y
 $O(n^2)$ worst average
if already sorted as input
(break while loop immediately) < >
best $O(n)$

What if we want to compare our own custom objects?

We have two options.

public interface Comparable<T>

This interface imposes a total ordering on the objects of each class that implements it. This ordering is referred to as the class's *natural ordering*, and the class's `compareTo` method is referred to as its *natural comparison method*.

public interface Comparator<T>

A comparison function, which imposes a *total ordering* on some collection of objects. Comparators can be passed to a sort method (such as `Collections.sort` or `Arrays.sort`) to allow precise control over the sort order. Comparators can also be used to control the order of certain data structures (such as `sorted sets` or `sorted maps`), or to provide an ordering for collections of objects that don't have a *natural ordering*.

The ordering imposed by a comparator `c` on a set of elements `S` is said to be *consistent with equals* if and only if `c.compare(e1, e2) == 0` has the same boolean value as `e1.equals(e2)` for every `e1` and `e2` in `S`.

→ need to define `public int compareTo(T otherObj)`

what if you're using
someone else's class?
(and cannot
modify)

→ need to define `public int compare(T obj1, T obj2)`

interfaces
↓
need to use
keyword
"implements"

implementing compareTo(Movie otherMovie)
within the **Movie** class so it can be **Comparable**.

```
class Movie implements Comparable<Movie> { // make sure to import java.util.*
    public String title;
    public int year;
    public double rating;

    public Movie(String title, int year, double rating) {
        this.title = title;
        this.year = year;
        this.rating = rating;
    }

    public int compareTo(Movie otherMovie) {
        if (rating < otherMovie.rating)
            return -1;
        else if (rating > otherMovie.rating)
            return 1;
        return 0;
    }

    public String toString() {
        return title + " (" + year + "), rating = " + rating;
    }
}
```

defined within class

implementing the `compare(Movie movie1, Movie movie2)`
outside the **Movie** class to create a **Comparator**.

→ make it easier to switch
between
orderings

comparator 1

```
class MovieYearComparator implements Comparator<Movie> { // make sure to import java.util.*
    public int compare(Movie movie1, Movie movie2) {
        if (movie1.year < movie2.year)
            return -1;
        else if (movie1.year > movie2.year)
            return 1;
        return 0;
    }
}
```

comparator 2

```
class MovieTitleLengthComparator implements Comparator<Movie> {
    public int compare(Movie movie1, Movie movie2) {
        if (movie1.title.length() < movie2.title.length())
            return -1;
        else if (movie1.title.length() > movie2.title.length())
            return 1;
        return 0;
    }
}
```

its own
class

...

// Somewhere else in the code (possibly a PSVM) ...

// Create a Comparator<T> object and pass it to sort:

Arrays.sort(movies, new MovieYearComparator()); // sort by year

Arrays.sort(movies, new MovieTitleLengthComparator()); // sort by title length

Sorting Algorithm #3 (Bucket Sort):

Main idea: put items in buckets, sort each bucket, re-assemble.

1. Set up some number of buckets k .
2. **Scatter** all n items into the appropriate bucket.
3. **Sort** each bucket.
4. **Gather** items from buckets into sorted array.

$\{21, 9, 7, 3, 19, 14, 6, 12, 23, 16\}$

max value: 23

$k = 5$
buckets

Notes:

- Does not require items to be comparable (unless using comparison-based sorting for each bucket).
- Works well if the input data is uniformly distributed (i.e. buckets evenly sized).
- **Disadvantage:** how to determine number of buckets k ? (need information about input data).
- Worst-case runtime: $\mathcal{O}(n^2)$.
- Average-case runtime: $\mathcal{O}(n + k)$.
- Not in-place, but stable.



sort each bucket



gather

$\{3, 6, 7, 9, 12, 14, 16, 19, 21, 23\}$



Sorting Algorithm #4 (Radix Sort):

max # digits = 3 = # passes = k

Main idea: similar to bucket sort, use digits to make buckets.

{ 170, 45, 75, 90, 2, 802, 2, 66 }

{ 170, 045, 075, 090, 002, 802, 002, 066 }

1. Pick a radix (base for each digit; we'll use 10).
2. For each digit d (starting from least significant digit):
 1. Make 10 empty buckets for this digit's possible values (0 - 9).
 2. Get the d^{th} digit of each item and put into the appropriate bucket.
 3. Go back through all buckets and put items from each bucket back into the original array.

pass 1: — — X

0	170, 090
1	
2	002, 802, 002
3	
4	
5	045, 075
6	066
7	
8	
9	

pass 2: — — X

0	002, 802, 002
1	
2	
3	
4	045
5	
6	066
7	170, 075
8	
9	090

pass 3: X — —

0	002, 002, 045, 066, 075, 090
1	170
2	
3	
4	
5	
6	
7	
8	802
9	

Notes:

- Does not require items to be comparable.
- Worst-case runtime: $\mathcal{O}(n \cdot k)$ (k is the maximum number of digits).
- Average-case runtime: $\mathcal{O}(n \cdot k)$.
- Not in-place, but stable.



See you tomorrow!

- We'll practice with implementing some of these sorting algorithms.
- Reminder that Noah ([go/noah](#)) and Smith ([go/smith](#)) have office hours throughout the week and the 201 Course Assistants have drop-in hours in the late afternoons/evenings ([go/cshelp](#)).
- Submit exit ticket 4R today.