CompSci 201, L14: Sorting

Announcements, Coming up

- Today, Wednesday 3/1
  - APT 5 (linked list problems) due
- Next Monday 3/6
  - Project P3: DNA (linked list project) due
- Next Wednesday 3/8
  - APT 6 (sorting problems) due
- Then…Spring Break!

Today's outline

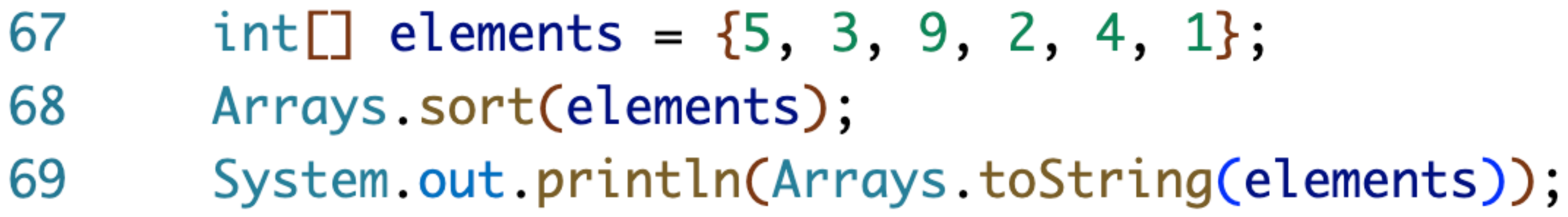
1. Sorting in Java: Comparing objects with `Comparable` and `Comparator`

2. Efficient sorting algorithms: recursive mergesort

Sorting in Java: Comparable, Comparator

Sorting like Java.util: Put elements of an Array/List in non-decreasing order

- `Arrays.sort` / `Collections.sort` are void – they sort the array/list passed as an argument.
- Default order is non-decreasing (least to greatest).



```
int[] elements = {5, 3, 9, 2, 4, 1};
Arrays.sort(elements);
System.out.println(Arrays.toString(elements));
```

- Prints `[1, 2, 3, 4, 5, 9]`

Java API Sort Algorithms

- `Collections.sort` (for a List)
- `Arrays.sort` (for an Array)

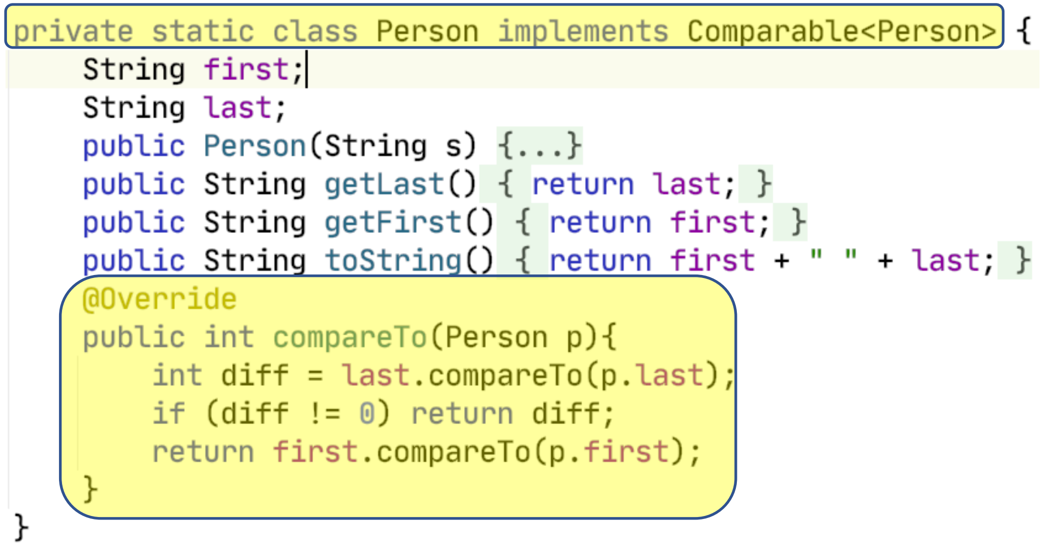
- Both O(N log(N)), *nearly* linear runtime complexity.
- Sorts in-place, mutates the input rather than return a new List/Array.
- Stable, does not reorder elements if not needed (e.g., if two elements are equal).

What can be compared and sorted in Java?

- Objects of a Class that implements `Comparable` interface. Has a `naturalOrder`.
- Requires implementing a `.compareTo()` method

Should return an int:

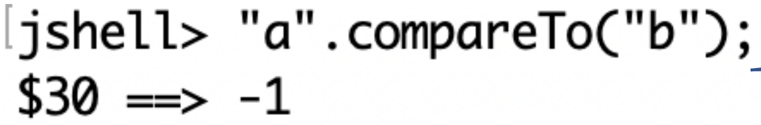
- < 0 if `this` comes before the parameter.
- 0 if `this` and the parameter are equal.
- > 0 if `this` comes after the parameter.



```
private static class Person implements Comparable<Person> {
    String first;
    String last;
    public Person(String s) {...}
    public String getLast() { return last; }
    public String getFirst() { return first; }
    public String toString() { return first + " " + last; }
    @Override
    public int compareTo(Person p){
        int diff = last.compareTo(p.last);
        if (diff != 0) return diff;
        return first.compareTo(p.first);
    }
}
```

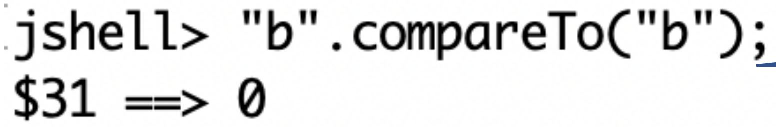
Strings are Comparable

- What is the equivalent of < for Strings?
- Use the `compareTo` method for the natural lexicographic (dictionary/sorted) ordering.



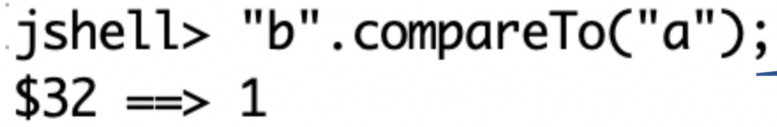
```
jshell> "a".compareTo("b");
$30 ==> -1
```

Negative for “less than”



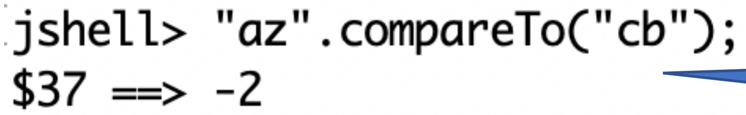
```
jshell> "b".compareTo("b");
$31 ==> 0
```

Zero for “equal”



```
jshell> "b".compareTo("a");
$32 ==> 1
```

Positive for “less than”

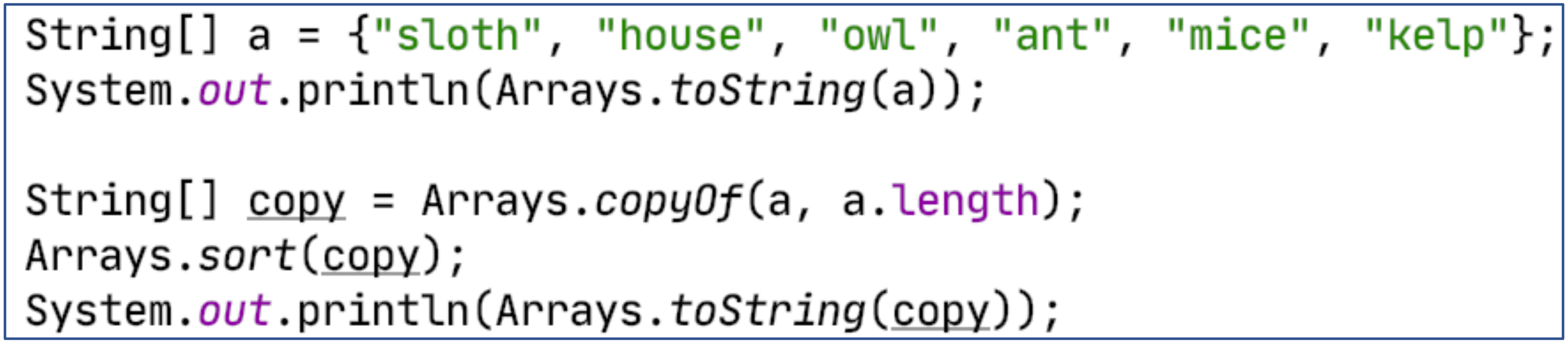


```
jshell> "az".compareTo("cb");
$37 ==> -2
```

Lexicographic, check first character, second if equal, third if still equal, …

Sorting Comparable objects by naturalOrder

```
[sloth, house, owl, ant, mice, kelp]
```



```
String[] a = {"sloth", "house", "owl", "ant", "mice", "kelp"};
System.out.println(Arrays.toString(a));

String[] copy = Arrays.copyOf(a, a.length);
Arrays.sort(copy);
System.out.println(Arrays.toString(copy));
```

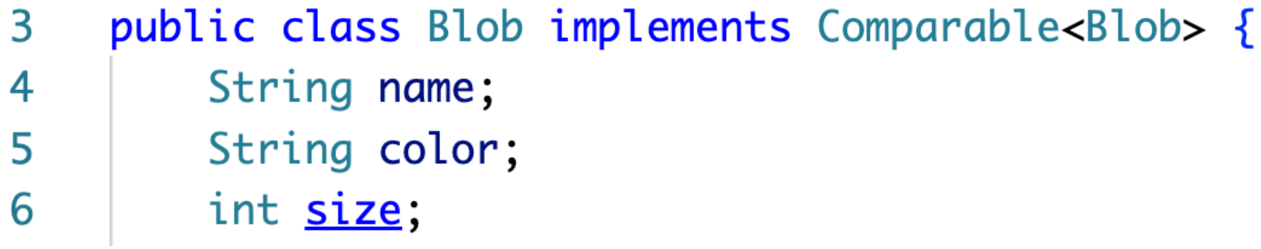
```
[ant, house, kelp, mice, owl, sloth]
```

- naturalOrder for Strings is lexicographic (alphabetical or dictionary order)

Comparable for other classes?

All Blob comparing code available here

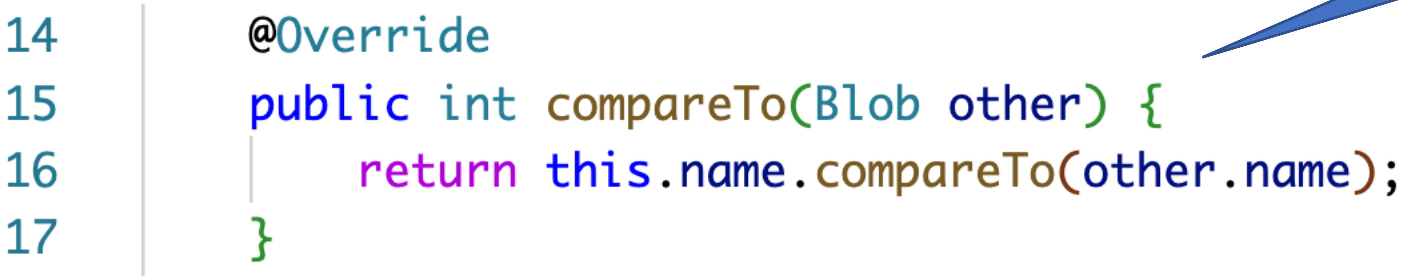
- Can implement `Comparable` interface when defining your own class.



```java
public class Blob implements Comparable<Blob> {
    String name;
    String color;
    int size;
```

- Must implement a `compareTo` method

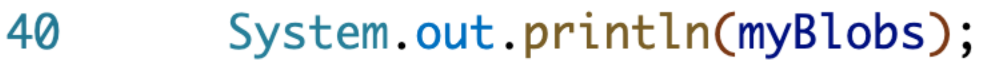
Compares blobs by their names



```java
    @Override
    public int compareTo(Blob other) {
        return this.name.compareTo(other.name);
    }
```

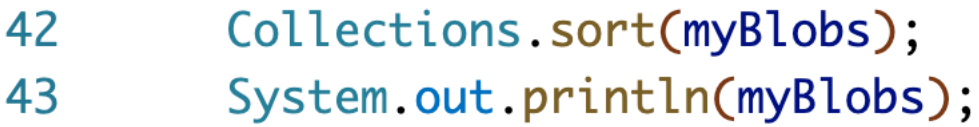
Sorting Comparable Objects

- Running code in a main method…



```
System.out.println(myBlobs);
```

Original: `[(bo, blue, 4), (al, red, 2), (cj, green, 1), (di, red, 4)]`



```
Collections.sort(myBlobs);
System.out.println(myBlobs);
```

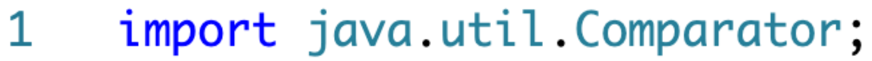
Sorted: `[(al, red, 2), (bo, blue, 4), (cj, green, 1), (di, red, 4)]`

- Formal guarantee: Element `e1` will come before `e2` (after sorting) if `e1.compareTo(e2) < 0`.

Defining a Comparator

- What if…
  - The class doesn’t implement Comparable?
  - Or you want to sort a different way?
- Create a helper class that implements the `Comparator` interface.
  - One method: `compare`: indicates how to compare two objects
- Then pass a Comparator object to your call to sort.

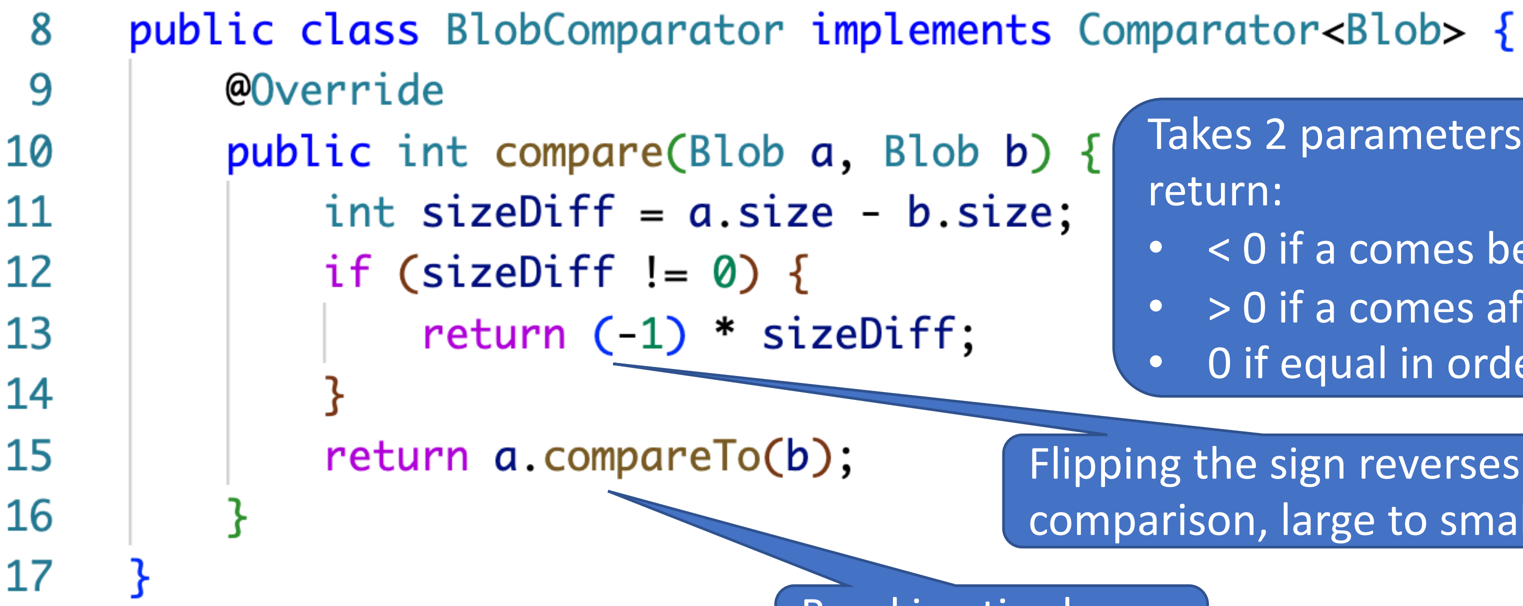
Defining a Comparator<Blob>



```
import java.util.Comparator;
```

Separate class:

- implements Comparator<TypeToCompare>,
- and implements a single method compare



```
public class BlobComparator implements Comparator<Blob> {
    @Override
    public int compare(Blob a, Blob b) {
        int sizeDiff = a.size - b.size;
        if (sizeDiff != 0) {
            return (-1) * sizeDiff;
        }
        return a.compareTo(b);
    }
}
```

Takes 2 parameters, Should return:

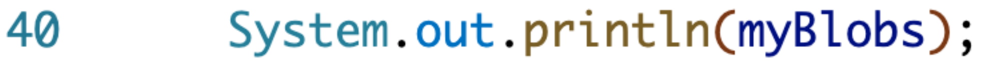
- < 0 if a comes before b,
- > 0 if a comes after b,
- 0 if equal in order

Flipping the sign reverses the comparison, large to small

Breaking ties by the natural order

Sorting with a Comparator

- Running code in a main method…

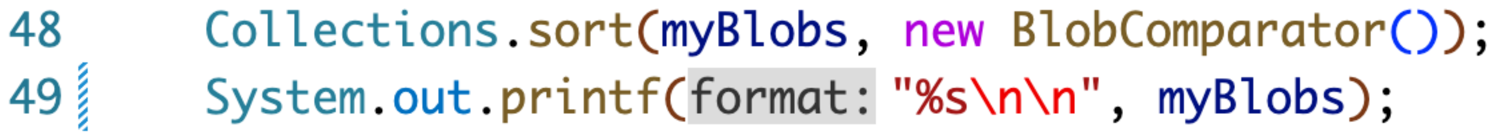


```
        System.out.println(myBlobs);
```

Original:

```
[(bo, blue, 4), (al, red, 2), (cj, green, 1), (di, red, 4)]
```

Create a `BlobComparator` object, pass it to the sort.



```
      Collections.sort(myBlobs, new BlobComparator());
      System.out.printf(format: "%s\n\n", myBlobs);
```

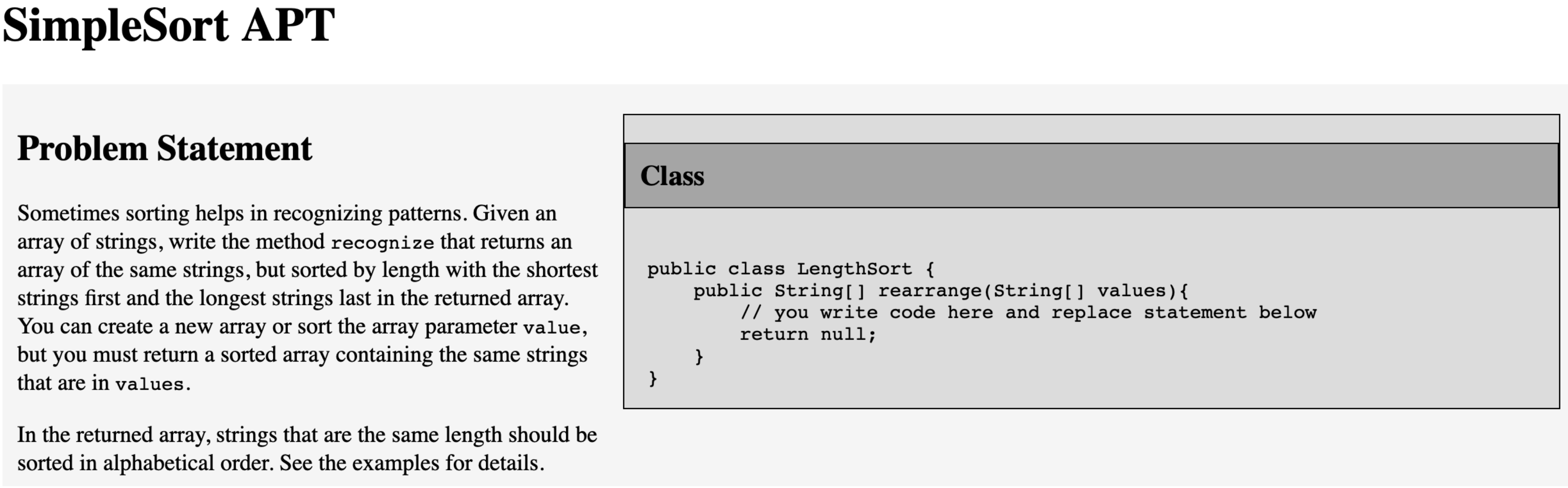
Sorted:

```
[(bo, blue, 4), (di, red, 4), (al, red, 2), (cj, green, 1)]
```

- Element `e1` will come before `e2` (after sorting) if `compare(e1, e2) < 0`.

Private Inner Comparator

- Can define a Comparator class as a private inner class if only used inside the class.
- Useful for APTs, here is an example:



SimpleSort APT

Problem Statement

Sometimes sorting helps in recognizing patterns. Given an array of strings, write the method `recognize` that returns an array of the same strings, but sorted by length with the shortest strings first and the longest strings last in the returned array. You can create a new array or sort the array parameter `value`, but you must return a sorted array containing the same strings that are in `values`.

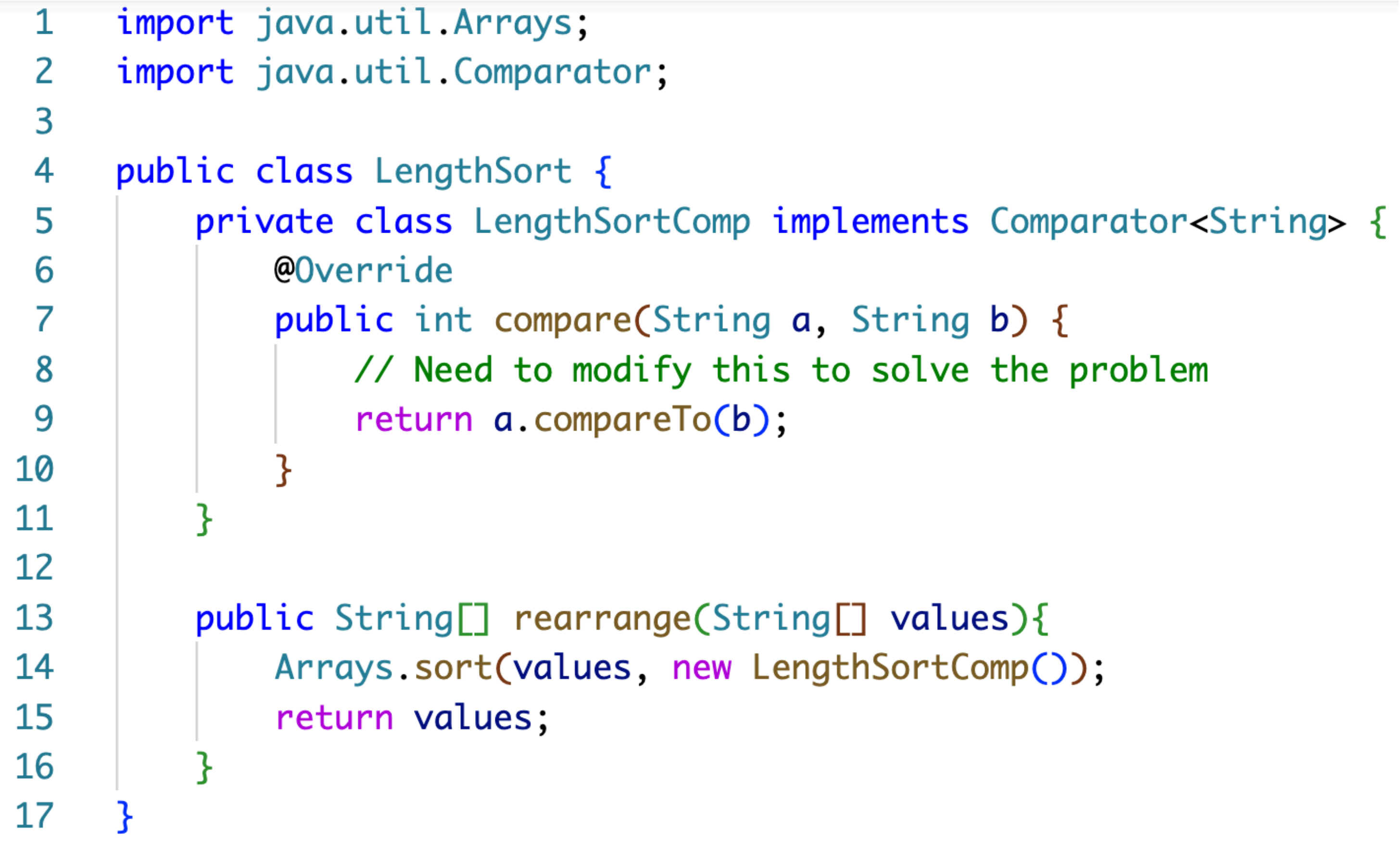
In the returned array, strings that are the same length should be sorted in alphabetical order. See the examples for details.

Class

```
public class LengthSort {
    public String[] rearrange(String[] values){
        // you write code here and replace statement below
        return null;
    }
}
```

Template for Solving LengthSort with a Private Inner Comparator

Can see this code here



```java
import java.util.Arrays;
import java.util.Comparator;

public class LengthSort {
    private class LengthSortComp implements Comparator<String> {
        @Override
        public int compare(String a, String b) {
            // Need to modify this to solve the problem
            return a.compareTo(b);
        }
    }

    public String[] rearrange(String[] values){
        Arrays.sort(values, new LengthSortComp());
        return values;
    }
}
```

`Comparable` vs. `Comparator`

- `Comparable a`: use `a.compareTo(b)`
  - What is method signature? One parameter
  - Method in class of which object a is an instance
  - a is `this`, b is a parameter

- `Comparator c`, use `c.compare(a,b)`
  - Method has two parameters
  - Part of `Comparator` (Java API link)
  - Returns an int:
    - < 0 (means a comes before b)
    - == 0 (means a equals b)
    - > 0 (means a comes after b)

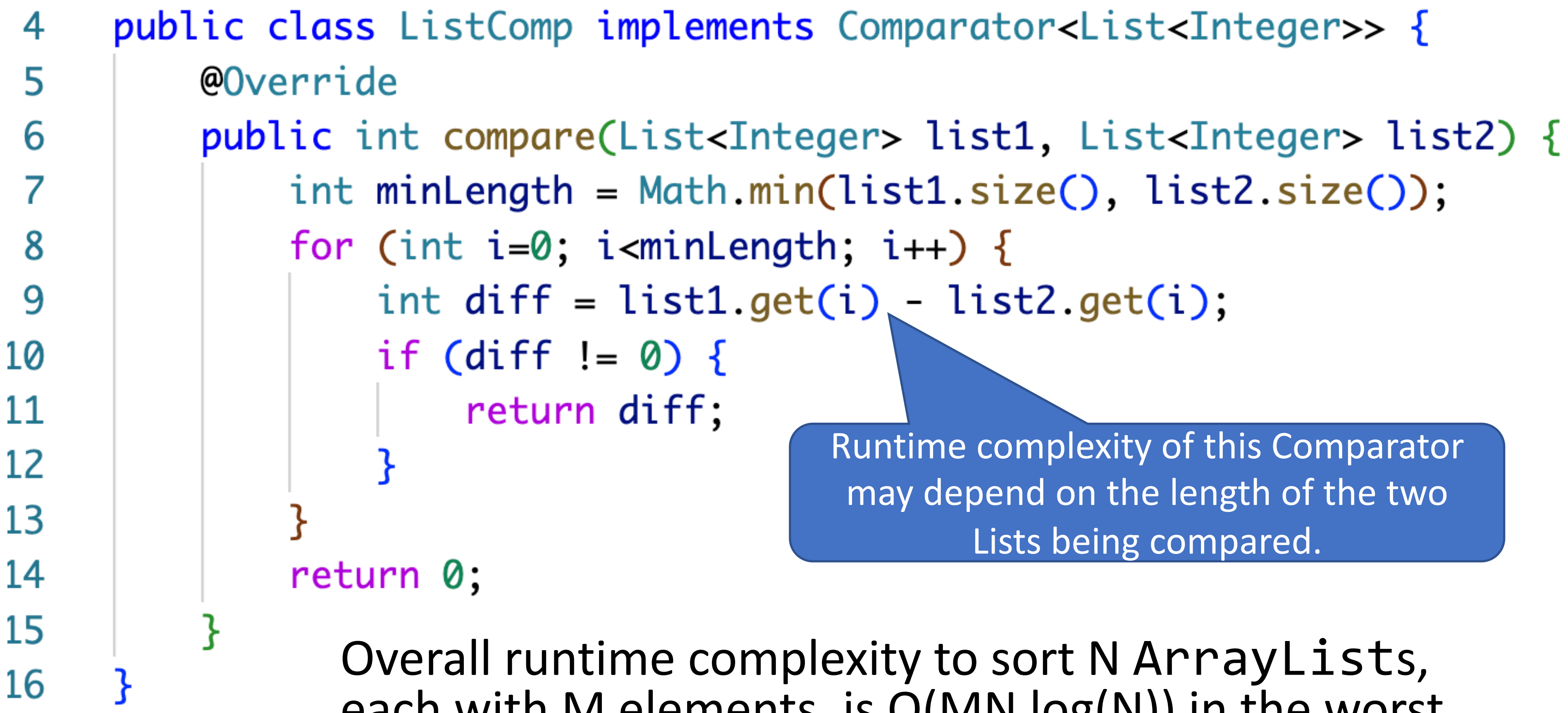
Runtime Complexity of Sort and Comparator?

- `Arrays.sort`, `Collections.sort`, call either `compareTo` (default) or `compare` (if you give a `Comparator`)…

- O(N log(N)) times, on an Array/List of N elements.

- Theoretical proof that this many comparisons is *necessary* for any comparison-based sorting.

When would C not be constant?



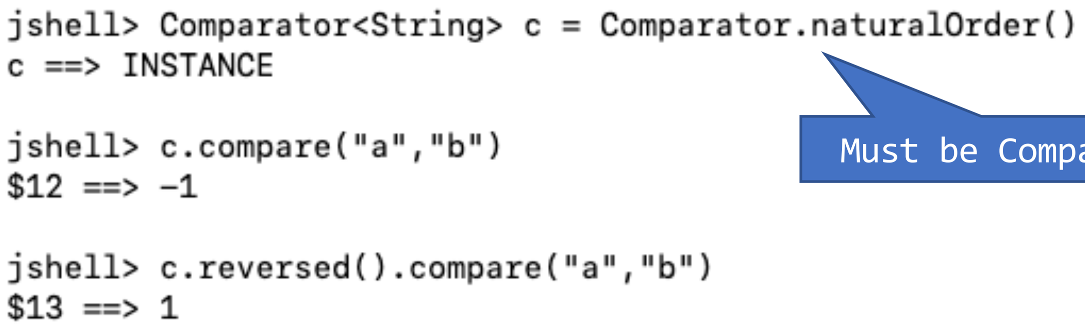
```java
public class ListComp implements Comparator<List<Integer>> {
    @Override
    public int compare(List<Integer> list1, List<Integer> list2) {
        int minLength = Math.min(list1.size(), list2.size());
        for (int i=0; i<minLength; i++) {
            int diff = list1.get(i) - list2.get(i);
            if (diff != 0) {
                return diff;
            }
        }
        return 0;
    }
}
```

Runtime complexity of this Comparator may depend on the length of the two Lists being compared.

Overall runtime complexity to sort N `ArrayLists`, each with M elements, is O(MN log(N)) in the worst case with this `Comparator`.

java.util.Comparator: Convenient Shorthands

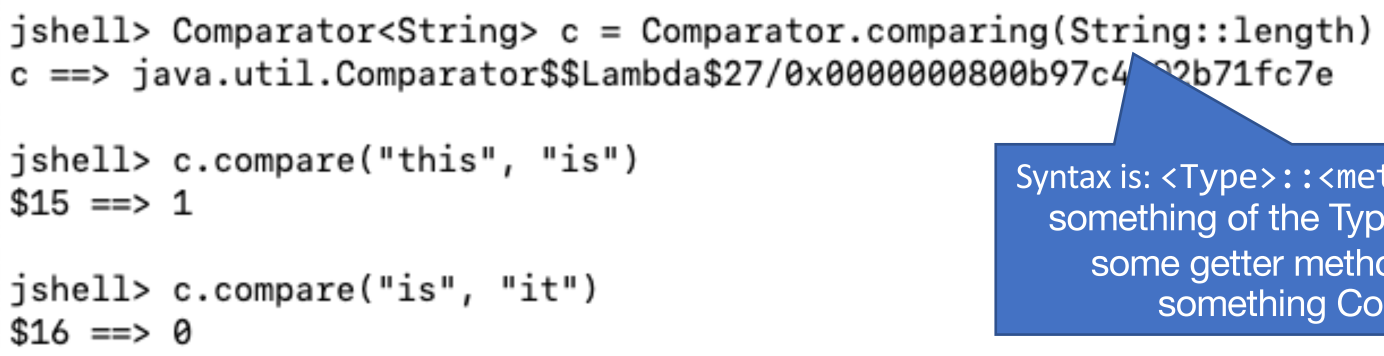
- `Comparator.naturalOrder` and `reversed()`



```
jshell> Comparator<String> c = Comparator.naturalOrder()
c ==> INSTANCE

jshell> c.compare("a","b")
$12 ==> -1

jshell> c.reversed().compare("a","b")
$13 ==> 1
```

Must be Comparable

- `Comparator.comparing`

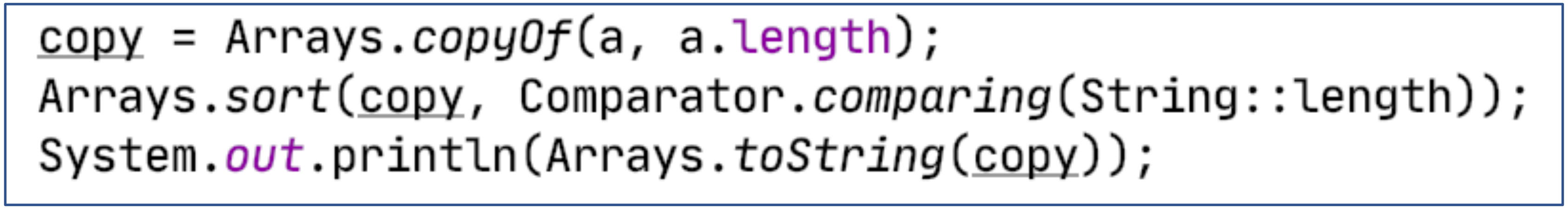


```
jshell> Comparator<String> c = Comparator.comparing(String::length)
c ==> java.util.Comparator$$Lambda$27/0x0000000800b97c40@2b71fc7e

jshell> c.compare("this", "is")
$15 ==> 1

jshell> c.compare("is", "it")
$16 ==> 0
```

Syntax is: `<Type>::<method name>` to sort something of the Type by the result of some getter method that returns something Comparable.

Using **Comparator** generating shorthands

[sloth, house, owl, ant, mice, kelp]



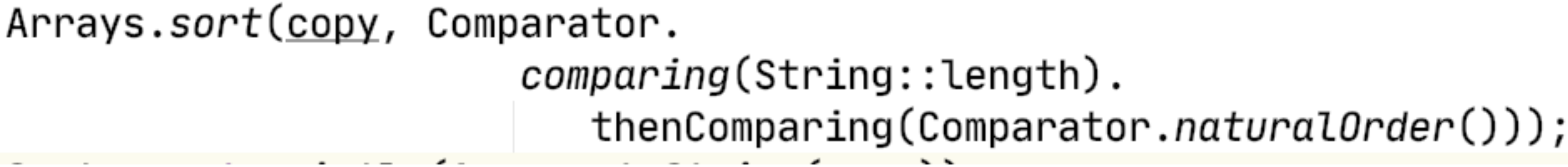
```
copy = Arrays.copyOf(a, a.length);
Arrays.sort(copy, Comparator.comparing(String::length));
System.out.println(Arrays.toString(copy));
```

[owl, ant, mice, kelp, sloth, house]

- Why does "owl" come before "ant"?
  - Stable sort respects order of equal keys

Using `.thenComparing` shorthand

`[sloth, house, owl, ant, mice, kelp]`



```
Arrays.sort(copy, Comparator.
                        comparing(String::length).
                           thenComparing(Comparator.naturalOrder()));
```

`[ant, owl, kelp, mice, house, sloth]`

- First compare by length
  - if same? Compare naturally

Comparator with “lambdas”

- Can also define a comparator with a “lambda” expression.

```
Integer[] nums = {2, 0, 1};

Comparator<Integer> comp = (a, b) -> (b-a);
```

Type we want to compare

Given an a and a b of that type…

`comp.compare(a,b)` should return this expression

`Arrays.sort(nums, comp);` `nums` is now `{ 2, 1, 0 }`

WOTO

Go to duke.is/8qrxr

Not graded for correctness, just participation.

Try to answer *without* looking back at slides and notes.

But do talk to your neighbors!

2

What is printed by the following line of code?

System.out.println("duke".compareTo("devils")); *

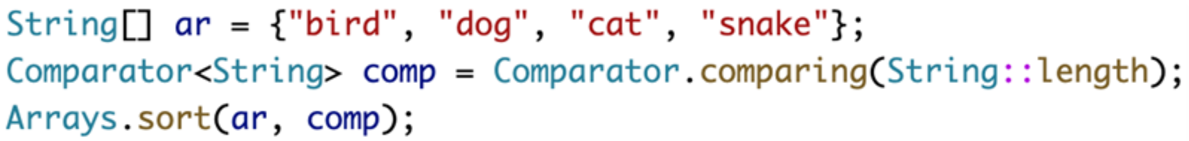
○ an integer less than 0

○ 0

○ an integer greater than 0

3

After sorting, ar will be... *



```
String[] ar = {"bird", "dog", "cat", "snake"};
Comparator<String> comp = Comparator.comparing(String::length);
Arrays.sort(ar, comp);
```

○ [dog, cat, bird, snake]

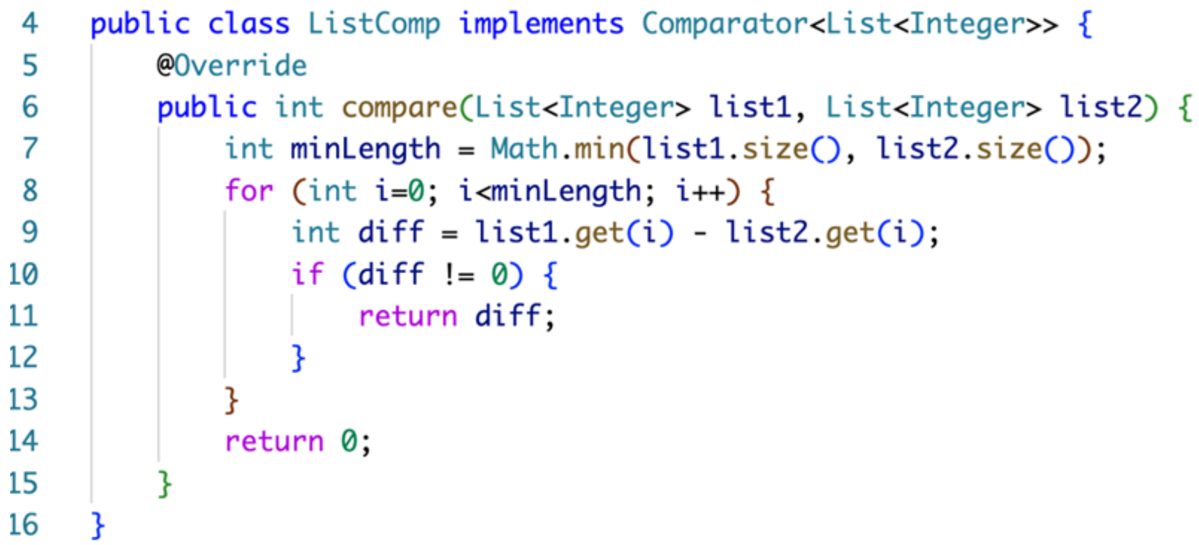
○ [cat, dog, bird, snake]

○ [snake, bird, cat, dog]

4

Suppose you have the following list of lists of integers:

[[2, 0, 1], [1, 0, 1], [1, 6]]. After sorting, the list would be ordered as... *



```
public class ListComp implements Comparator<List<Integer>> {
    @Override
    public int compare(List<Integer> list1, List<Integer> list2) {
        int minLength = Math.min(list1.size(), list2.size());
        for (int i=0; i<minLength; i++) {
            int diff = list1.get(i) - list2.get(i);
            if (diff != 0) {
                return diff;
            }
        }
        return 0;
    }
}
```

○ [[1, 6], [1, 0, 1], [2, 0, 1]]

○ [[1, 6], [2, 0, 1], [1, 0, 1]]

○ [[1, 0, 1], [1, 6], [2, 0, 1]]

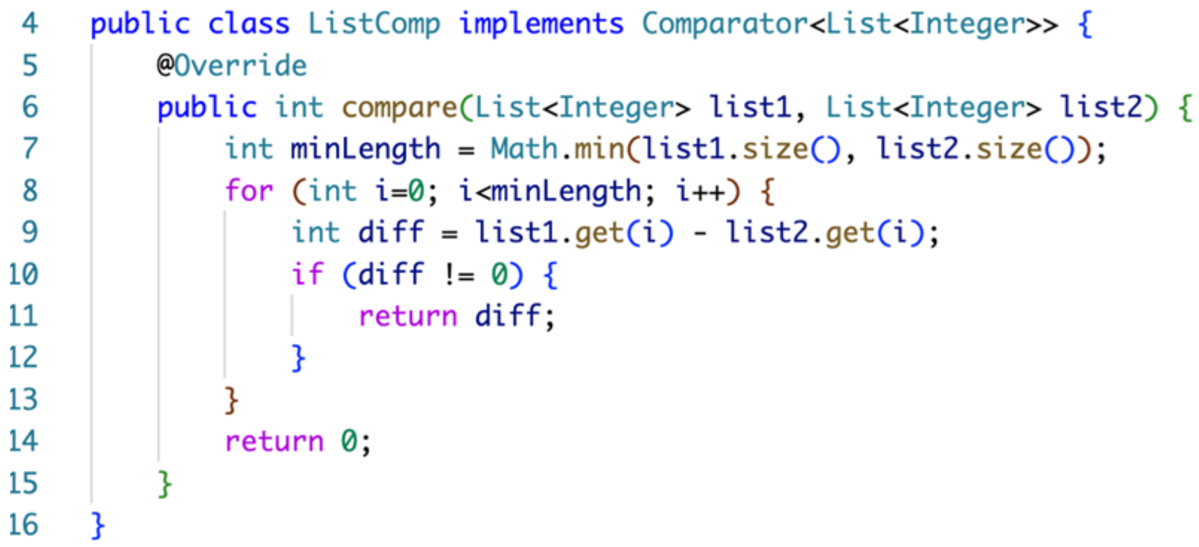
○ [[1, 0, 1], [2, 0, 1], [1, 6]]

○ [[2, 0, 1], [1, 6], [1, 0, 1]]

○ [[2, 0, 1], [1, 0, 1], [1, 6]]

5

Suppose you have an ArrayList myLists of N ArrayLists, each of size at most M. The worst-case runtime complexity to compare any two elements of myLists would be... *



```
public class ListComp implements Comparator<List<Integer>> {
    @Override
    public int compare(List<Integer> list1, List<Integer> list2) {
        int minLength = Math.min(list1.size(), list2.size());
        for (int i=0; i<minLength; i++) {
            int diff = list1.get(i) - list2.get(i);
            if (diff != 0) {
                return diff;
            }
        }
        return 0;
    }
}
```

○ O(N)

○ O(M)

○ O(N log(N))

○ O(M log(M))

○ O(NM log(N))

○ O(NM log(M))

6

Given an Array of N Strings, each of length at most M, the worst case runtime complexity to sort the Array with java.util.Arrays.sort is... *

○ O(N)

○ O(M)

○ O(N log(N))

○ O(M log(M))

○ O(NM log(N))

○ O(NM log(M))

This content is neither created nor endorsed by Microsoft. The data you submit will be sent to the form owner.

Microsoft Forms

Efficient sorting algorithms: recursive mergesort

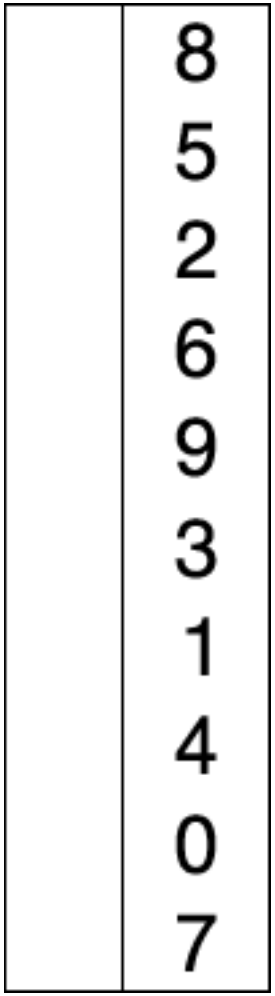
See example implementations here

Selection Sort with a Loop Invariant

- Loop invariant: On iteration `i`, the first `i` elements are the smallest `i` elements in sorted order.

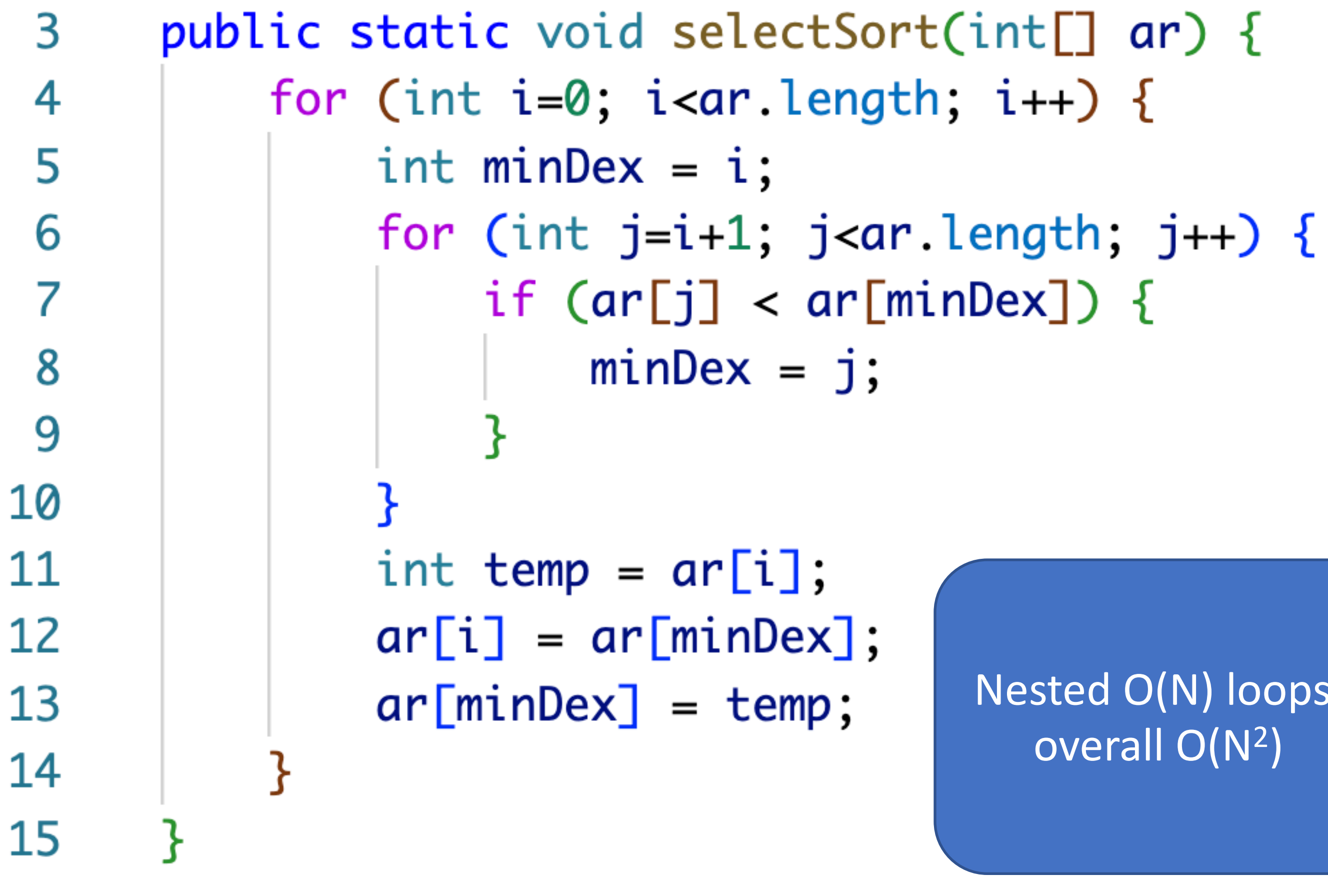
- On iteration `i`…
  - Find the smallest element from index `i` onward
    - (By loop invariant, must be the next smallest element)
  - Swap that with the element at index `i`

- Algorithm is called *Selection Sort.*



By Joestape89, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=3330231

Selection Sort Code and Runtime



```java
public static void selectSort(int[] ar) {
    for (int i=0; i<ar.length; i++) {
        int minDex = i;
        for (int j=i+1; j<ar.length; j++) {
            if (ar[j] < ar[minDex]) {
                minDex = j;
            }
        }
        int temp = ar[i];
        ar[i] = ar[minDex];
        ar[minDex] = temp;
    }
}
```

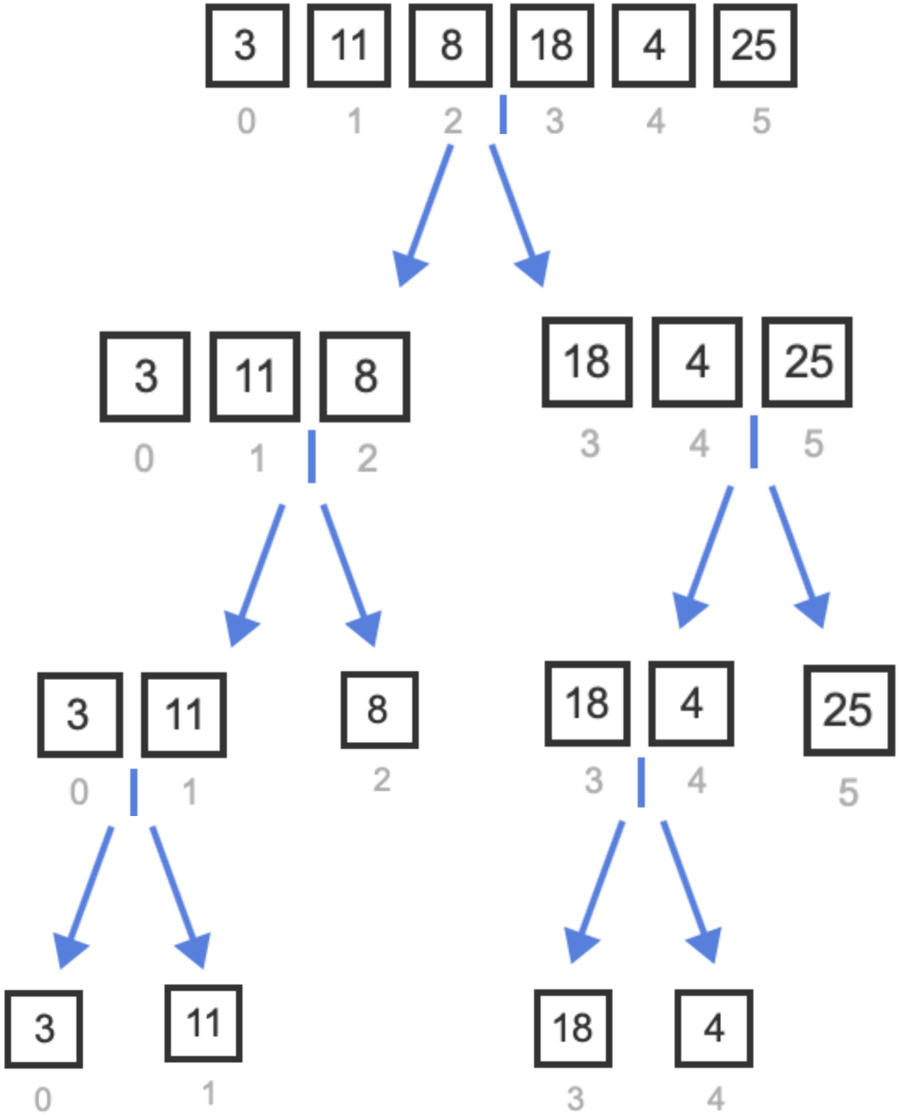
Nested O(N) loops, overall O(N^2)

	8
	5
	2
	6
	9
	3
	1
	4
	0
	7

Mergesort

High level idea:

- Base case: size 1
  - Return list
- Recursive case:
  - Mergesort(first half)
  - Mergesort(second half)
  - …



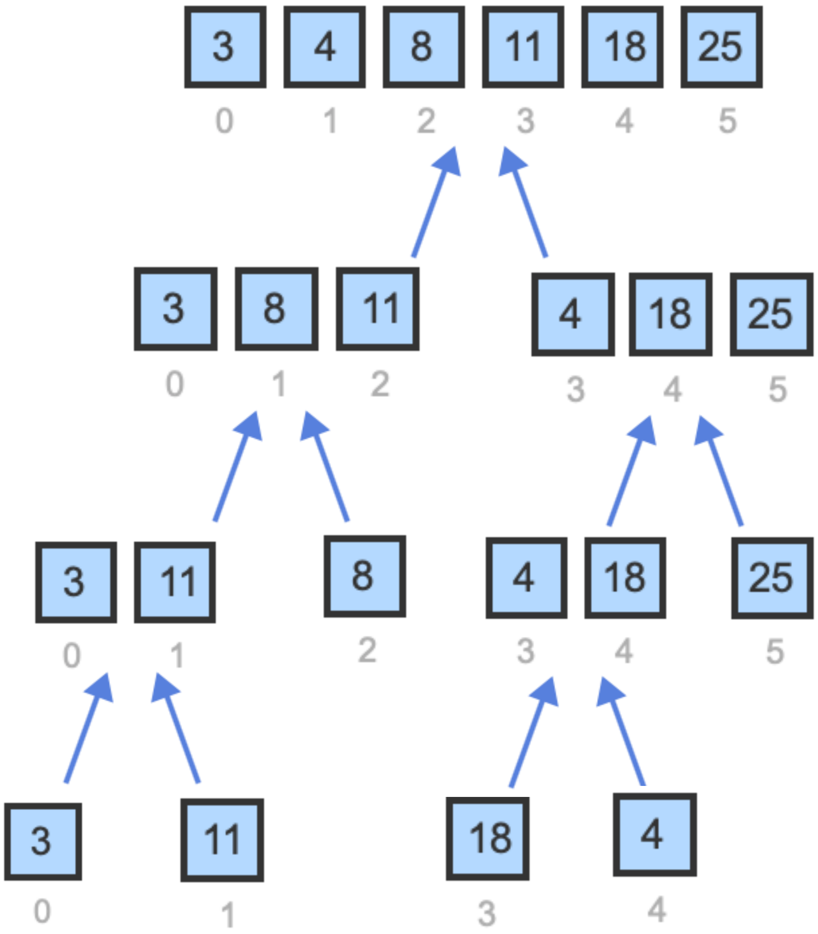
Zybook

Mergesort

High level idea:

- Base case: size 1
  - Return list
- Recursive case:
  - Mergesort(first half)
  - Mergesort(second half)
  - Merge the sorted halves
  - Return sorted

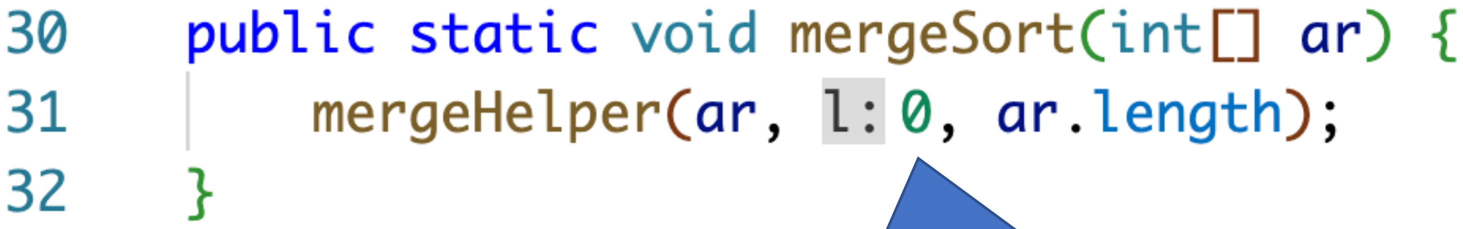
Helper method



Zybook

Mergesort recursive wrapper

- A recursive wrapper method:
  - Is the top-level method a user would call,
  - Is not itself recursive, but makes the initial call to a recursive method,
  - Allows recursive helper method to have additional parameters.

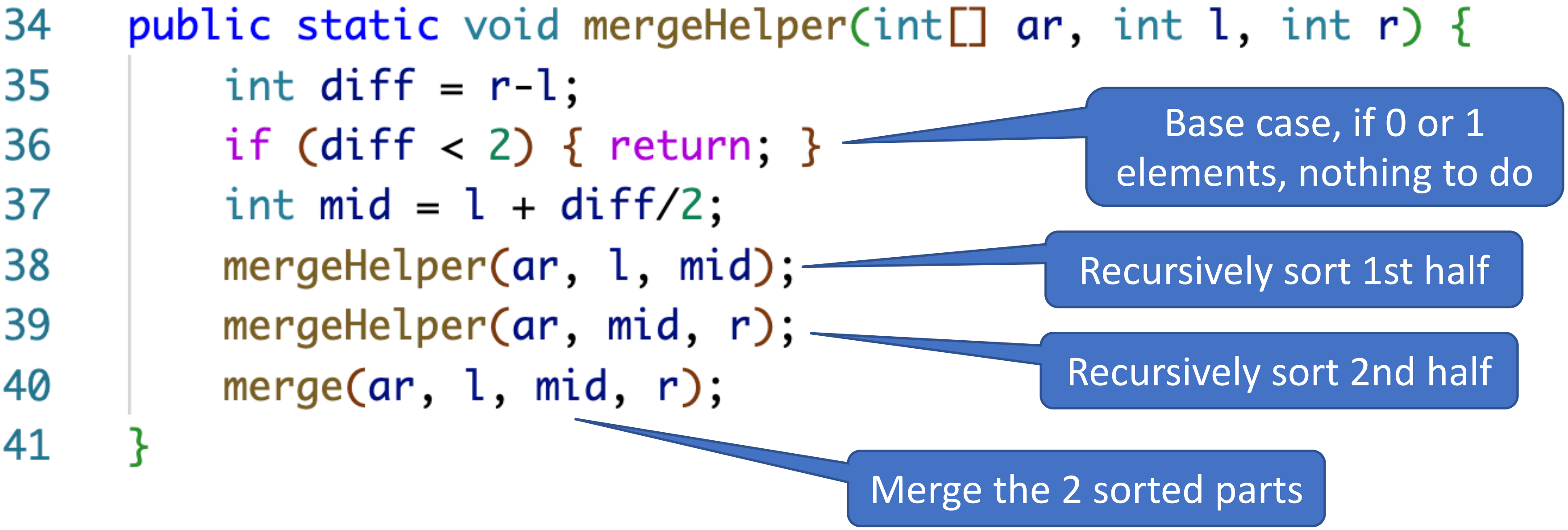


```
public static void mergeSort(int[] ar) {
    mergeHelper(ar, l: 0, ar.length);
}
```

Want to specify a left and right boundary of the subarray for each recursive call to sort

Mergesort recursive method

- Should sort everything in **`ar`** starting at index **`l`** and up to (but not including) index **`r`**.



```
public static void mergeHelper(int[] ar, int l, int r) {
    int diff = r-l;
    if (diff < 2) { return; }
    int mid = l + diff/2;
    mergeHelper(ar, l, mid);
    mergeHelper(ar, mid, r);
    merge(ar, l, mid, r);
}
```

Base case, if 0 or 1 elements, nothing to do

Recursively sort 1st half

Recursively sort 2nd half

Merge the 2 sorted parts