# SWEN 601 Software Construction

*Sorts & Complexity*

Today’s Quiz Password: BinRec

## Activity: Accept the GitHub Classroom Assignment

You will be asked to accept a new assignment at the start of nearly every class.

You should get used to accepting the assignment and starting your new project right after you finish your quiz each day.

Your instructor has provided a GitHub classroom invitation. You should be able to find it under “Homework” on MyCourses.

1. Click the GitHub classroom invitation.
2. Assuming that you have already linked your GitHub account with your name in the class roster, you should be prompted to accept the assignment. Do so.
3. Once the repository is created, copy the URL.
4. Clone the repository to your local file system. As before, the repository will be empty.
5. Create a new IntelliJ Project inside the repository, and push it to GitHub.
6. Create a package named “`activities`” in your `src` folder.
7. You are now ready to begin today’s activities!

## Warm Up: Sorted?

Some of the arrays that we will be sorting will be very large. We will need a way to determine whether or not the array is sorted without printing it out and visually inspecting it.

1. Create a new class in the `activities` package called `SortUtilities`.
2. Add a `public static` method `sorted(int[])` that returns `true` if the array parameter is sorted, and `false` otherwise.
    a. Hint: compare the value at each index *i* to the value at index *i-1*.

# Warm Up: Swapping Values

Many of the algorithms that we write will need to swap values between indexes in the same array.

1. In the `SortUtilities` class, add a `public static` function called `swap(int[] array, int a, int b)` that swaps the values in indexes a and b.

# java.util.Random

- The `java.util.Random` class is a ***pseudorandom*** number generator.
  - The numbers that it generates are based on an initial ***seed***.
  - Any instance of the class created with the same ***seed*** will generate the same pseudorandom numbers in the same sequence.
  - If created without a seed, the constructor chooses a seed.
- The `Random` class provides several useful methods:
  - `nextDouble()` - returns a random in the range 0 to 1.
  - `nextBoolean()` - returns a random boolean; effectively a “coin flip.”
  - `nextInt(int bound)` - returns a random integer between 0 and `bound-1`.
- We will use this class to make pseudorandom test data for our sorting algorithms.
  - But we will use a seed, so that the pseudorandom numbers will be predictable every time that we run our tests.

```
import java.util.Random;
```

You will need to `import` `Random` from the `java.util` package in order to use it.

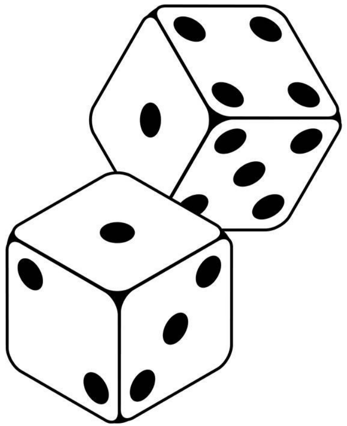
```
Random RNG = new Random();
```

Creating an instance of `Random` without a seed will make it generate numbers unpredictably.

```
Random RNG = new Random(1);
```

But using a seed guarantees that it generates the same sequence of predictable numbers.

# Warm Up: Random Data



We will need to create arrays filled with interesting, unsorted values so that we can test our sorting functions.

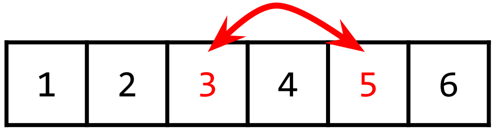
1. In the `SortUtilities` class, the following field declaration to the top of the class.

```
private static final Random RNG = new Random(1);
```

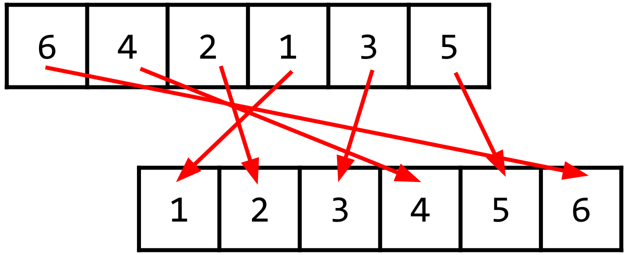
2. add a `public static` function `makeArray(int size)` that returns an array of the specified size.
   a. Use the `nextInt(int bound)` method on the RNG field to set the value in every index in the array to a number between 0 and size.

# Sorting Vocabulary

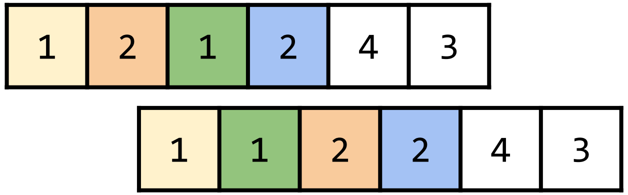
- Before we begin examining sorting algorithms, it is important to understand some common sorting terms.
  - ***In Place*** refers to a sort that moves elements around inside of the array. The original order of the elements is lost. A sort that is ***not*** in-place will create a sorted copy of the array.
  - ***Partition*** refers to dividing an array into two or more parts. For an *in place* sort, this usually means keeping track of an index that indicates the end of one partition and the beginning of the next.
  - ***Stability*** refers to whether or not the relative order of equal elements will change as a result of the sort. If the relative order is maintained, the sort is ***stable***.
  - ***Natural order*** refers to the desired ordering of the elements when the sort is finished. For example, will the sort arrange integers from largest to smallest? Or smallest to largest?



An ***in place*** sort swaps values in the same array.



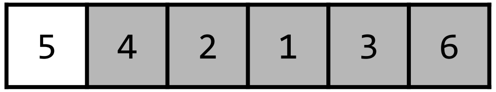
A sort that is not in place makes a sorted ***copy*** of the original array.



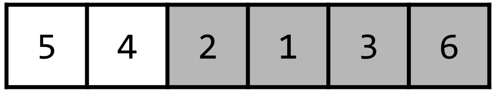
***Stability*** guarantees that the relative order of *equal* elements will not change during the sort.

# Insertion Sort

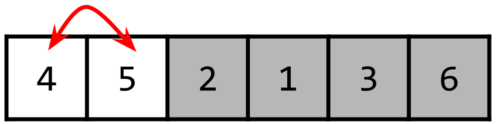
- The concept of ***Insertion Sort*** is simple.
  - Divide the array into two ***partitions***: a ***sorted*** partition and an ***unsorted*** partition.
    - At first the sorted partition contains only the first element in the array.
    - The unsorted partition includes the rest of the elements.
  - Move the ***leftmost*** element from the unsorted partition into the sorted partition.
    - If the newly added element is out of place, ***swap*** it with its neighbor to the left.
    - ***Continue*** swapping until it is in place.
  - Repeat until all of the elements have been added to the sorted partition.
- Insertion sort is easily implemented as an ***in place*** sort.
  - An ***index*** is used to mark the first element in the unsorted partition.



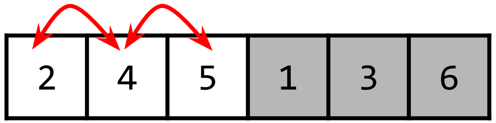
Begin by partitioning the array into ***sorted*** and ***unsorted*** partitions.



Each ***iteration*** moves one element into the sorted partition.

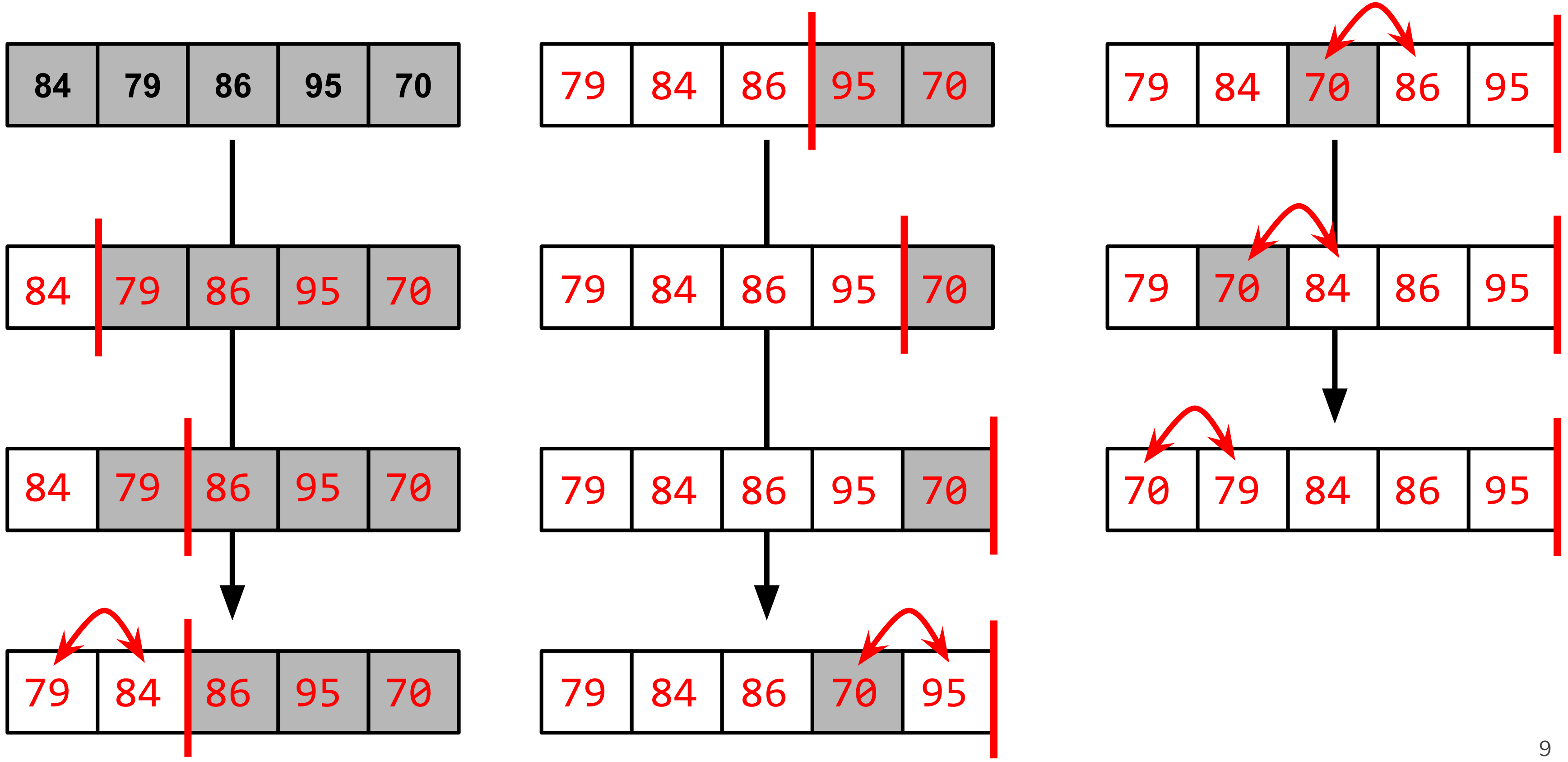


If it is out of place, it will need to be ***swapped*** with is neighbor.

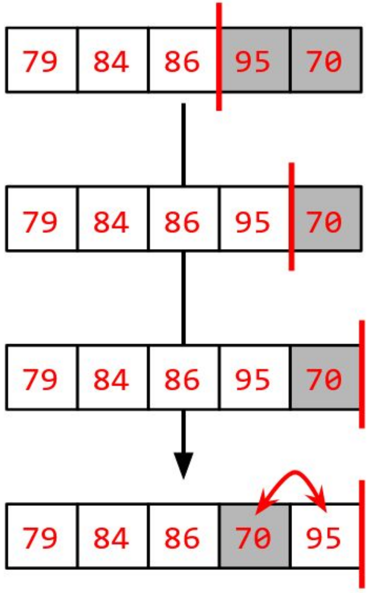


Sometimes, multiple swaps are needed...

## Insertion Sort Visual Example



# Activity: Implement Insertion Sort



Let's implement our first sort!

1. In the `activities` package, create a new class called `Sorts`.
2. Add a `public static` method `insertionSort` that performs an ***in place*** sort on an array passed in as a parameter.
3. Begin by setting the index that marks the start of the *unsorted* partition to 0.
4. You will need two loops:
    a. One that moves the partition to the next index.
    b. One that swaps the out of place values with their neighbors to the left.
        i. *Hint*: You may use your swap function.

# Insertion Sort: Complexity Analysis

Let's take a look at the ***worst case*** complexity for insertion sort: the array is sorted in ***reverse order***.

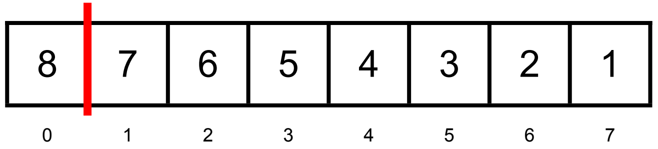
- Given ***0 ≤ k < n***, the element at index ***k*** is swapped ***k*** times.
- The average number of swaps is therefore:

$$\sum_{k=0}^{n} \frac{k}{n} \cong \tfrac{1}{2} n$$

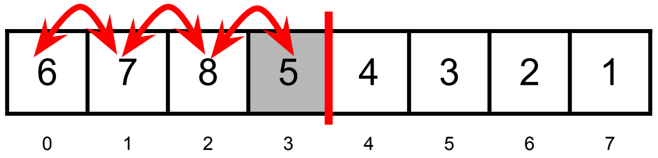
- Given an array of size ***n***, the average of ***½ n*** swaps needs to be performed ***n times***. Therefore:

$$n * \tfrac{1}{2} n = \tfrac{1}{2} n^2 = O(n^2)$$

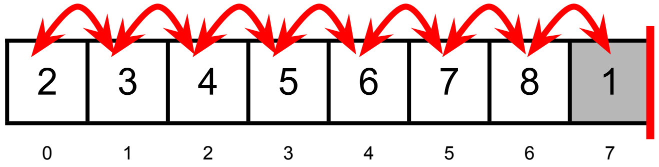
When 8 (index 0) is added to the sorted list, it is swapped ***0*** times.



When 5 (index 3) is added to the sorted list, it is swapped ***3*** times.

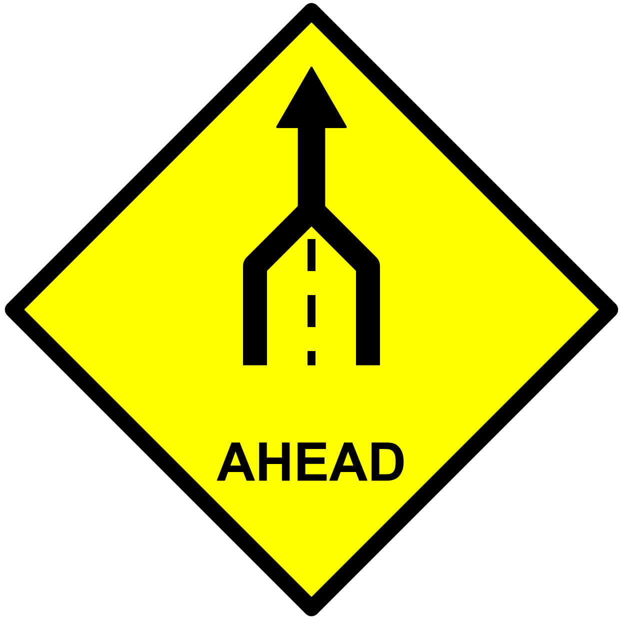


When 1 (index 7) is added to the sorted list, it is swapped ***7*** times.



## A More Efficient Sort: *Merge Sort*

- There are *lots* of different sorting algorithms. Some perform better than others.
- Next, we are going to discuss ***Merge Sort***.
- The concept of merge sort is a little more complex than insertion sort.
  - ***Divide*** the array into ***sub-arrays***, each of about ***length ½ n***.
    - Continue dividing the sub-arrays until the lists contain only a ***single element***.
  - ***Merge*** the sub-arrays back together in ***sorted order***.
  - Continue ***merging*** until ***all*** of the sub-arrays are merged back together.



Merge sort copies elements from one list to another, and is therefore ***not*** an in place sort.

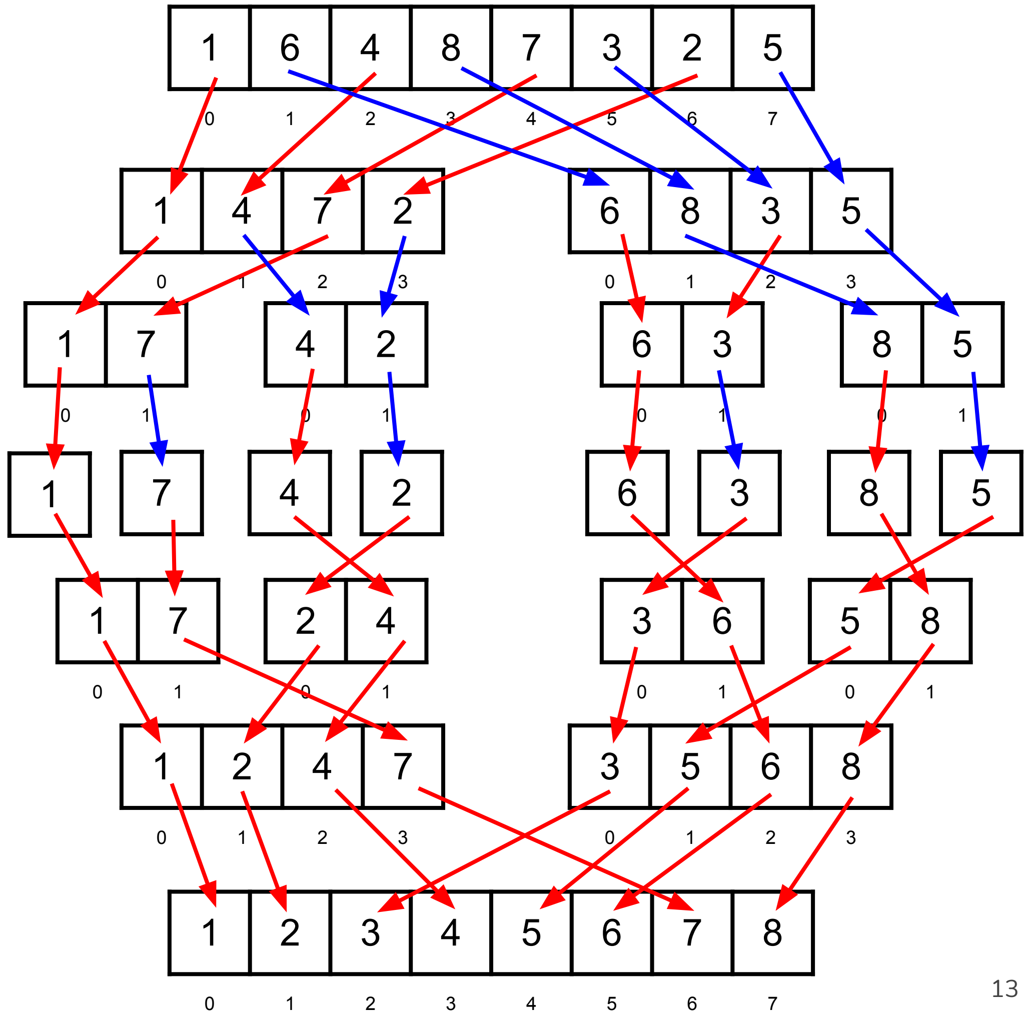
# Merge Sort Visual Example

The first question you must answer when implementing merge sort is *how* to divide the array into sub-arrays.

One option is to take the ***even-odd*** approach. We copy all of the values at an ***even index*** into one sub-array...

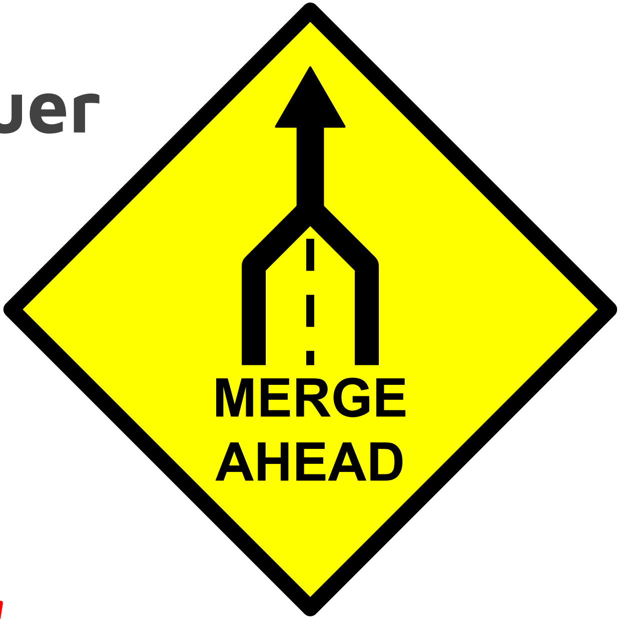
...and all of the values at an ***odd index*** into the other sub-array.

Note that even/odd does **not** refer to the ***value***; it refers to the ***index***. Imagine an array with ***only*** odd values in it!



# Merge Sort: Divide and Conquer

- Like Binary Search, Merge Sort is a ***divide and conquer*** algorithm:
  - ***Divide*** an array of length ***n*** into two sublists of length ***n/2***.
  - ***Recursively sort*** the two sublists (this is the ***conquer*** step).
  - ***Merge*** the two sorted sublists back together in ***sorted order***.
- To understand the time complexity of Merge Sort, we need to analyze the complexity of each step in the ***divide and conquer*** algorithm.



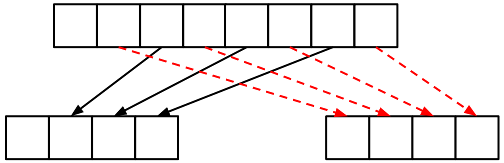
The premise of merge sort is that it's easier to sort smaller lists than larger ones.

So a large list is first divided into smaller ones.

# Activity: Merge Sort: Divide

The first step in the merge sort algorithm requires that an array of n elements be divided into two sub-arrays with ***n/2*** elements in each array.

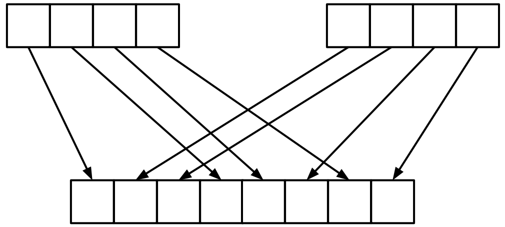
1. Add a `public static divide(int[] array)` method to your `SortUtilities` class.
    a. Create two arrays: one for evens (length/2), and another for odds (length - evens.length).
    b. Use a for loop to:
        i. Copy the values at even numbered indexes into the evens array.
        ii. Copy the values at odd numbered indexes into the odds array.
    c. Return both arrays as a two-dimensional array.



Q: What is the complexity of this method?

A: It must iterate ***n*** times, and therefore has a complexity of **O(n)**.

# Activity: Merge Sort: Merge



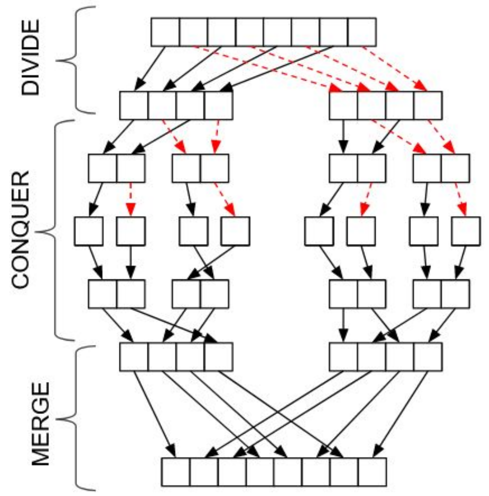
Q: What is the complexity of this method?

A: It must iterate ***n*** times, and therefore has a complexity of **O(n)**.

The last step in the merge sort algorithm requires that two arrays of n/2 elements be merged into a single array with n elements.

1. Add a `public static merge(int[] a, int[] b)` method to your `SortUtilities` class.
   a. Create one array that is the total length of both a and b.
   b. Use a for loop to merge the two arrays together.
   c. You will need an index for each array to keep track of the next value in the array.
      i. Compare the values at the index in each array and choose the smaller.
      ii. Increment that index.
   d. If one array ends first, copy the remaining elements from the other array.
   e. Return the merged array.

# Activity: Merge Sort

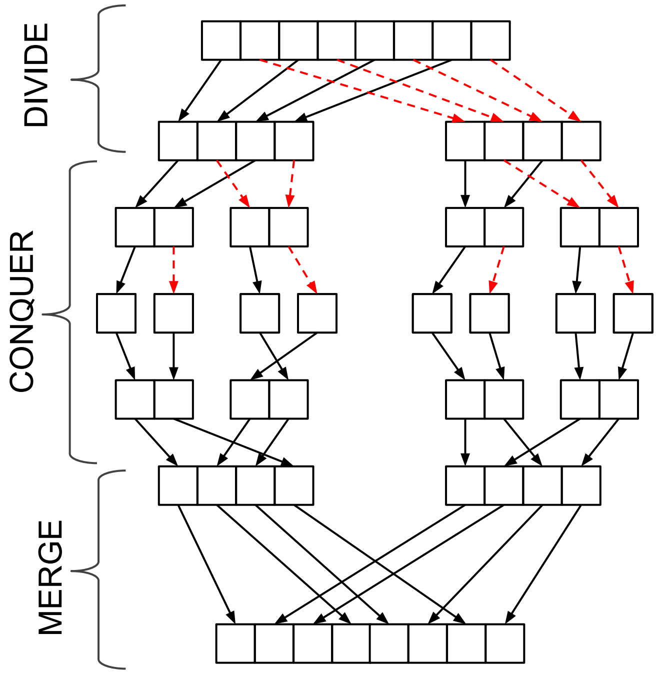


Let's implement merge sort!

1. Add a `public static` method `mergeSort(int[] array)` to your `Sorts` class.
   a. If the `array` length is less than 2, `return` the array.
   b. Call your `divide` method to divide the array in half.
   c. Recursively call `mergeSort` method on each half.
   d. Use your `merge` method to merge the arrays back together.
   e. Return the merged array!

# Merge Sort Complexity

- As mentioned previously, merge sort is a ***divide and conquer*** algorithm.
  - Divide: **O(n)** (iterate over the array of length **n**)
  - Merge: **O(n)** (iterate over 2 arrays with a total length of **n**).
- The key to the complexity of merge sort is the number of times the steps need to be repeated.
  - Q: How many times does an array of length **n** need to be divided before the parts are length < 2?
  - A: $\log_2 n$.
- If we need to perform an O(n) operation $\log_2 n$ times, then the total complexity of the divide operation is $\boldsymbol{O(n\log_2 n)}$.
- The total complexity of the merge operation is the same: we need to perform the O(n) merge operation $\log_2 n$ times: $\boldsymbol{O(n\log_2 n)}$.
- Therefore the total complexity is $\boldsymbol{O(2n\log_2 n)}$, but as n approaches infinity, we discard the scalar (2) and so the complexity is just $\boldsymbol{O(n\log_2 n)}$.



Merge sort always performs the ***same number*** of divides and merges regardless of the order in the lists.

Therefore its best, average, and worst case complexity are all the same: $O(n \log_2 n)$.

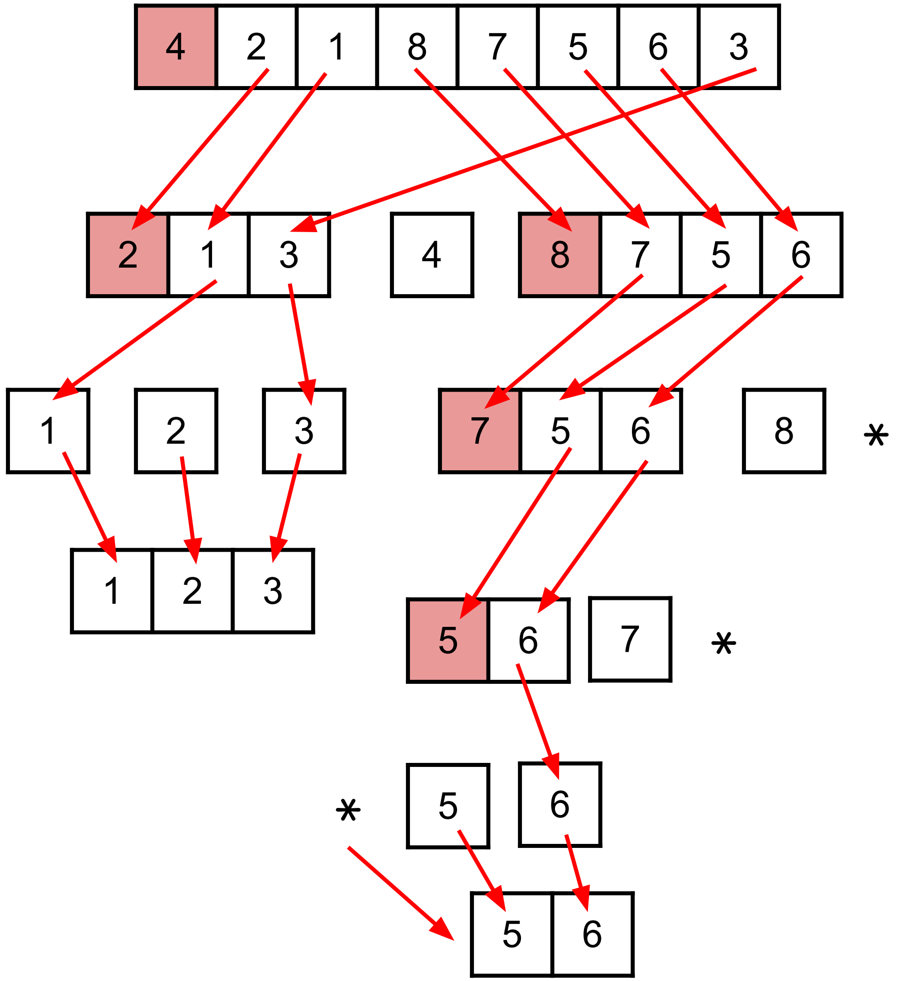
## Another Fast Sort: Quicksort

- Next we are going to discuss another fast sort: ***Quicksort***.
- Quicksort is also a ***fast*** sorting algorithm.
    - Choose a ***pivot*** value at some index. We'll use the value at ***index 0***.
    - Create three new arrays:
        - The ***less*** array will contain all of the values that are ***less than the pivot***.
        - The ***equal*** array will contain all of the values that are ***equal to the pivot***.
        - The ***greater*** array will contain all of the values that are ***greater than the pivot***.
    - ***Recursively sort*** the ***less*** and ***greater*** arrays.
        - There is no need to sort the ***equal*** array because all of the elements are equal!
    - ***Concatenate*** the three arrays back together: ***less + equal + greater***.

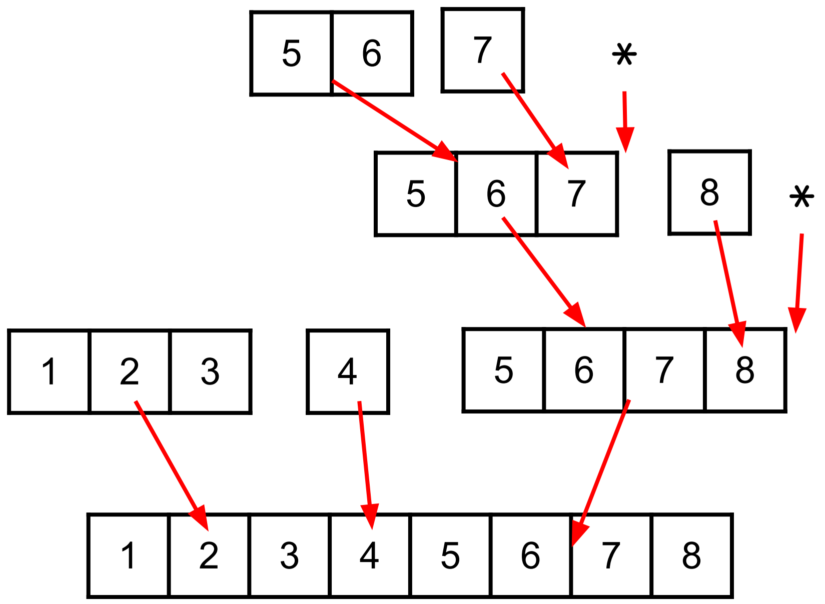
***Quicksort*** can be done **in place**, but our version is not implemented that way.

## Quicksort Visual Example I

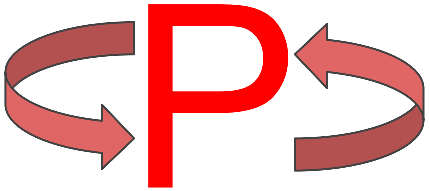
The pivot that you choose can greatly affect the performance of Quicksort. We will always choose the value at ***index 0***.



# Quicksort Visual Example II



# Quicksort: Complexity Analysis

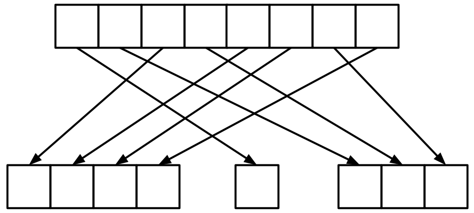


- Merge sort performs the same number of splits and merges regardless of how sorted or unsorted the data in the original array is.
- Quicksort is different. The performance of quicksort is impacted by two different factors:
  - The choice of the pivot.
  - The configuration of the data in the unsorted array.
- Like Merge Sort, Quicksort is a divide and conquer algorithm:
  - ***Divide*** data into ***less***, ***equal***, and ***greater*** arrays.
  - ***Recursively sort*** (conquer) the less and more arrays.
  - ***Concatenate*** the arrays back together.
- Let's take a look at the complexity of each step.

As with Merge Sort, the premise of Quicksort is that it's easier to sort smaller lists.

But unlike Merge Sort, the size of ***less*** and ***greater*** lists may vary greatly depending on the pivot choice and how sorted the data already is.

## Quicksort: Divide



- The first step in the quicksort algorithm requires that a ***pivot*** be chosen. Our algorithm always chooses the value at ***index 0***.
- We then create three arrays: ***less***, ***equal***, and ***greater***.
- The algorithm iterates over the unsorted array, and copies the elements into one of the three new arrays.
    - If the element is ***less than*** the pivot, it as added to ***less***.
    - If the element is ***greater than*** the pivot, it is added to ***greater***.
    - Otherwise, the element is added to ***equal***.
- This requires a loop that iterates ***once*** over the unsorted array, and therefore has a time complexity of **O(n)**.

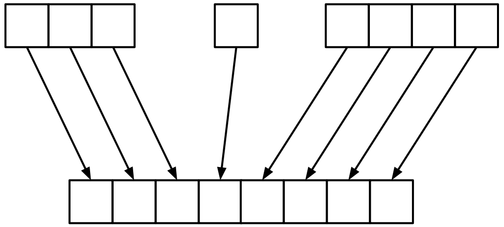
This step will always be **O(n)** regardless of the pivot or the arrangement of the data in the array.

However, depending on the pivot's relationship to the rest of the data, the lengths of the ***less***, ***equal***, and ***greater*** arrays may greatly vary.

Imagine, for example, that the pivot is the smallest (or largest!) value in the array...

# Quicksort: Concatenate

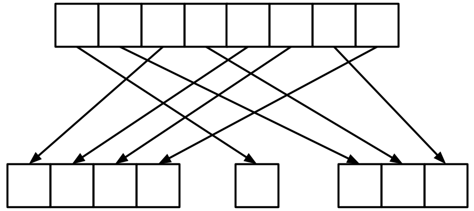
- The last step in the Quicksort algorithm requires that we concatenate the sorted ***less***, ***equal***, and ***greater*** arrays together to create the final, sorted copy of the unsorted list.
    - Let ***l*** be the length of ***less***.
    - Let ***e*** be the length of ***equal***.
    - Let ***g*** be the length of ***greater***.
    - We know that ***l + e + g*** must be ***n***.
- Array concatenation requires:
    - Creating a new array.
    - Copying the ***n*** elements from the ***less***, ***equal***, and ***greater*** arrays into the new array.
    - Copying ***n*** elements is an **O(n)** operation.
- Therefore, concatenating three arrays that contain a total of ***n*** elements together is an **O(n)** operation.



Again, this final step will always be **O(n)**, regardless of the lengths of the individual arrays.

An array of length ***n*** must be allocated and ***n*** elements copied from the other arrays.

## Quicksort: Recursively Sort (Best Case)



- As with Merge Sort, the tricky part of analyzing the complexity of Quicksort is the ***recursively*** ***sort*** (***conquer***) step.
- What if the number of elements in ***less*** and ***greater*** were always about the same (***½ n***)?
  - That is to say that, after the divide step, about ***half*** of the elements ended up in ***less***, and the other ***half*** ended up in ***greater***.
  - This means that, each time we divided an array, it would be cut in ***half***, meaning we'd need to divide about **O($lg_2$n)** times to get to arrays of length 1.
  - Then we'd need to concatenate the lists back together (**O(n)**).
  - This sounds just like Merge Sort (and it is).
  - The complexity for this operation would be **O(n*$log_2$n)**. Great!

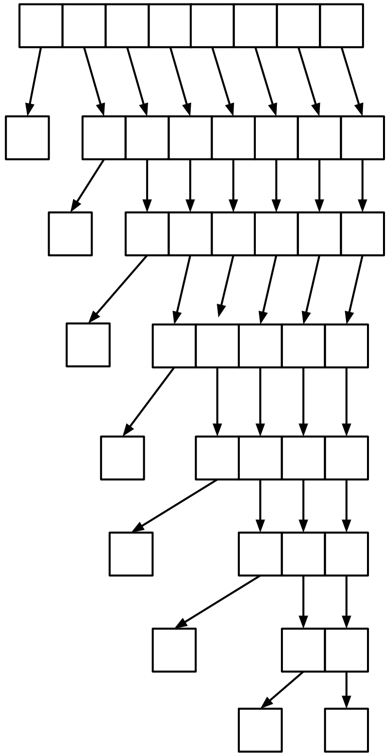
The performance of the divide and concatenate steps is always **O(n)** for a total of **O(2n)**.

So the performance of Quicksort depends on the conquer step.

In the best case, this is **O($log_2$n)** meaning the total complexity is **O(2n*$log_2$n)** or just **O(n*$log_2$n)**. Just like merge sort!

So what is the performance in the ***worst case***?

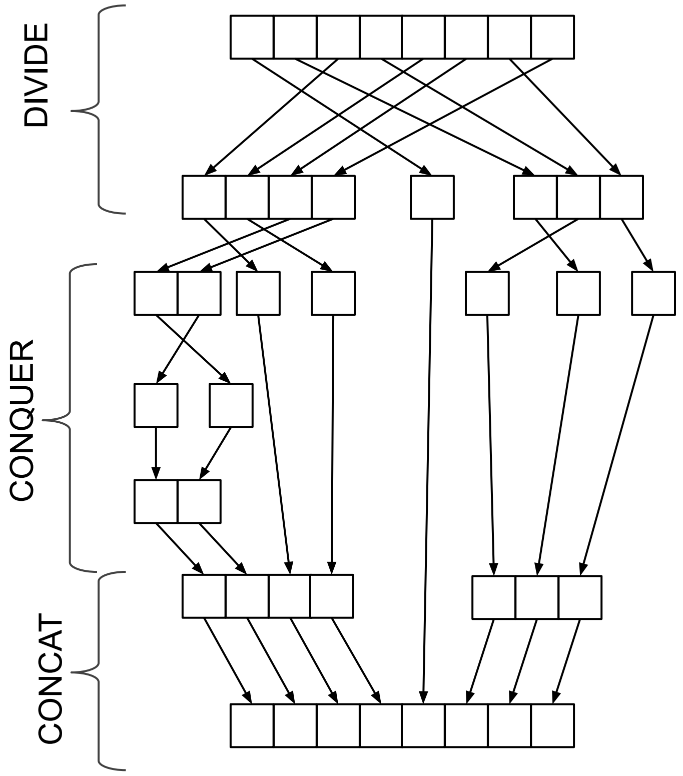
## Quicksort: Recursively Sort (Worst Case)



- What if every time a pivot was chosen, ***all*** of the elements ended up in ***either less*** or ***more***?
    - In other words, the pivot is the smallest or the largest value in the array.
    - One list would be ***length n-1*** and the other would be ***length 0***.
- How many times would the array with ***n*** elements in it need to be divided like this to get to a sub-array of length 1?
    - ***n*** times.
- The divide and merge operations (**O(2n)**) would still need to be performed for each sub-array.
- This means that an **O(2n)** operation needs to be performed ***n*** times, and the total complexity would therefore be **O(2n*n)** or **(On$^2$)**!

# Quicksort Complexity

- Like Merge Sort, Quicksort is a ***divide and conquer*** algorithm. The complexity is the sum of the complexities needed for each of the three steps.
  - Divide: **O(n)**
  - Conquer (recursively sort): ranges from **O(log$_2$n)** to **O(n)**.
  - Merge: **O(n)**
- Therefore, the best case performance of Quicksort is **O(2n*log$_2$n)** or **O(n*log$_2$n)** as n approaches infinity.
- And the worst case performance is **O(2n * n)** or **O(n$^2$)**.
- Thankfully the average case is **O(n*log$_2$n)**.



The worst case for Quicksort happens rarely; and so Quicksort routinely outperforms Merge Sort.