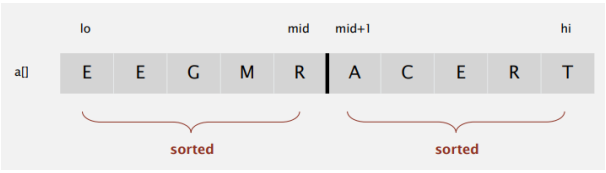
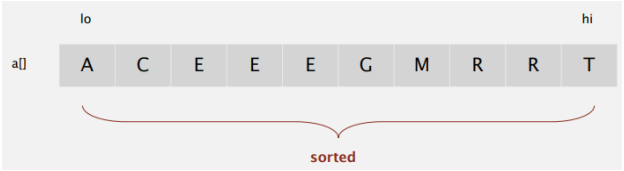
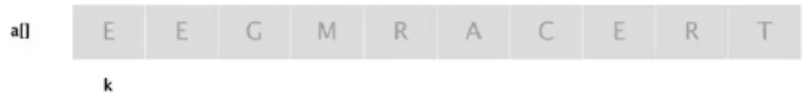
Mergesort (Week 3):

- Mergesort Concept/Introduction
  - Mergesort is a Java sort for objects whereas quicksort is a Java sort for primitives.
  - Basic plan
    - Divide the array into two halves.
    - Recursively sort each half.
    - Merge two halves.
  - Basic merging (explaining what merging is)
    - Goal
      - Given two sorted subarrays a[lo] to a[mid] and a[mid+1] to a[hi], replace the two sorted subarrays with the sorted subarray a[lo] to a[hi].



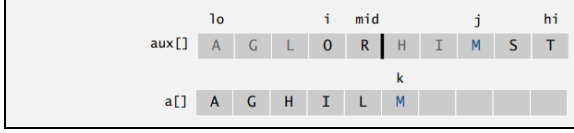


    - Method (abstract in-place merge demo)
      - 1. Create an auxiliary array
      - 2. Copy all elements to the auxiliary array
      - 3. Copy elements back into the main array but in sorted order.
        - To do this we keep track of 3 indices
          - The index on the left half of the array is i
          - The index on the right half of the array is j
          - The index on the main array is k



        - For each k index in the main array, you compare i and j and copy in the smaller value, then you increment the counter depending on which half of the array it came from (e.g., if i < j, copy value at j and increment j). Then you increment k so we move on to the next value for the main array.
        - If both values are the same, just take i's value and increment i.
        - Once i or j get to their max index, just take the rest of the other's subarray since the subarrays are already sorted.
    - Java implementation of merging

```
private static void merge(Comparable[] a, Comparable[] aux, int lo, int mid, int hi)
{
   assert isSorted(a, lo, mid);    // precondition: a[lo..mid]   sorted
   assert isSorted(a, mid+1, hi);  // precondition: a[mid+1..hi] sorted

   for (int k = lo; k <= hi; k++)                                          copy
      aux[k] = a[k];

   int i = lo, j = mid+1;
   for (int k = lo; k <= hi; k++)                                          merge
   {
      if      (i > mid)              a[k] = aux[j++];
      else if (j > hi)               a[k] = aux[i++];
      else if (less(aux[j], aux[i])) a[k] = aux[j++];
      else                           a[k] = aux[i++];
   }

   assert isSorted(a, lo, hi);     // postcondition: a[lo..hi] sorted
}
```



- First for-loop copies all the elements from the main array into the auxiliary array.
- Then we set i to the beginning of the first subarray and j to the beginning of the second subarray.
- The second for-loop runs through the auxiliary array and compares the values between i and j then sends whichever back to the main array at index k. It also makes sure that once one subarray is out of elements it adds the rest of the other subarray to the main array.
- Note that we use the assert command for the preconditions that the subarrays of the main array are both sorted and for the postcondition that makes sure the main array is sorted at the end of the method.

- Mergesort
  - Now taking what we learned from basic merging, we apply recursion and overloading to create the mergesort sorting method.

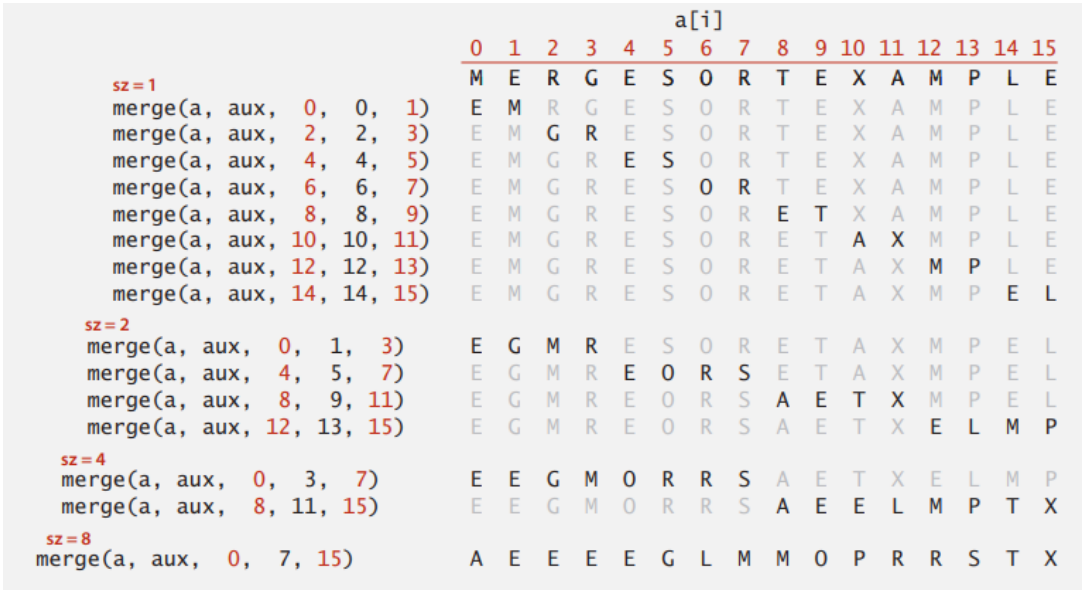
```
public class Merge
{
   private static void merge(...)
   {  /* as before */  }

   private static void sort(Comparable[] a, Comparable[] aux, int lo, int hi)
   {
      if (hi <= lo) return;
      int mid = lo + (hi - lo) / 2;
      sort(a, aux, lo, mid);
      sort(a, aux, mid+1, hi);
      merge(a, aux, lo, mid, hi);
   }

   public static void sort(Comparable[] a)
   {
      aux = new Comparable[a.length];
      sort(a, aux, 0, a.length - 1);
   }
}
```

- We use the same merge method from above.
- Then we create a new method sort(Comparable[] a) that takes an array of elements that have implemented the Comparable interface in their class. This method creates an auxiliary array and calls the sort(Comparable[] a, Comparable[] aux, int lo, int hi) method using the main array, the auxiliary array that was just created, the first/lowest index (0) and the last/highest index (mainArray.length-1).
- The sort(Comparable[] a, Comparable[] aux, int lo, int hi) method is where the actual sorting occurs.
    - The if statement ensures that the inputs for int lo and int hi make sense (hi can not be lower than lo).
    - Then we create a new int called mid which is the index at the middle of the array.
    - After that, we recursively call sort() twice.
        - The first one is to sort the first half of the array. (lo to mid)
        - The second one is to sort the second half of the array. (mid+1 to hi)
    - Since the halves of the array get sorted in-place, we then merge the two, now sorted, halves of the array back together using the merge() method.
- Recursion explanation
    - As mentioned above, we call sort(Comparable[] a, Comparable[] aux, int lo, int hi) twice inside the same method sort(Comparable[] a, Comparable[] aux, int lo, int hi).
    - The reason this works is that each time the method is called, it will keep dividing the array into halves until you get an array of size 1.
    - When you have an array of size 1, the if statement returns the array immediately as the value for hi and lo are the same (both are 0 since it is

the only index), in this case, since the array is of size 1, it is sorted no matter what which allows us to call the merge() method.
        - Now you call the merge() method on the two subarrays and it merges the subarray of size 1 with another subarray of size 1 (the other half that got divided), so you will now get a sorted subarray of size 2 (since the merge() method combines two sorted subarrays). Now the sorted subarray of size 2 will get merged with another sorted subarray of size 2 (the other half that got divided) and you will then get a sorted subarray of size 4, and so on until the entire subarray is sorted.
        - The same works for odd-sized arrays, it's just that the right "halves" would have one more element than the left "halves".
    - Empirical analysis of mergesort
        - Running time estimates:
            - Laptop executes 10^8 compares/second.
            - Supercomputer executes 10^{12} compares/second.

	insertion sort (N^2)			mergesort (N log N)		
computer	thousand	million	billion	thousand	million	billion
home	instant	2.8 hours	317 years	instant	1 second	18 min
super	instant	1 second	1 week	instant	instant	instant

        - Bottom line. Good algorithms are better than supercomputers.
    - Mathematical analysis of mergesort
        - Proposition
            - Mergesort uses at most NlgN compares and 6NlgN array accesses to sort any array of size N.
    - Note that there are practical improvements to mergesort (refer to slides).
- Bottom-up Mergesort
    - Bottom-up
        - Bottom-up means that there is no recursion.
        - Basic idea is to think of the array as being a combination of many little subarrays of size 1 (since size 1 means it is sorted already), this skips the splitting phase in the recursive method.
        - Next we use the merge() method on two subarrays of size 1 all the way to the end of the array. Now you have an array with multiple sorted subarrays of size 2. You continue, 1 2 4 8…, until eventually you will merge two halves of the array into one fully sorted array.



- Java implementation

```
public class MergeBU
{
   private static void merge(...)
   {  /* as before */  }

   public static void sort(Comparable[] a)
   {
      int N = a.length;
      Comparable[] aux = new Comparable[N];
      for (int sz = 1; sz < N; sz = sz+sz)
         for (int lo = 0; lo < N-sz; lo += sz+sz)
            merge(a, aux, lo, lo+sz-1, Math.min(lo+sz+sz-1, N-1));
   }
}
```

- You start off with getting the total length of the array you need to sort.
- Then you create an auxiliary array of the same size as the main array.
- The first for-loop will start with subarrays of size 1 and keep doubling until the size of the subarrays equals the size of the main array (which means it is sorted).
    - The inner for-loop will merge two subarrays of a certain size (given by the outer for-loop) all the way through the array. So if you start off with an array of 8 pairs of 2-sized sorted subarrays, you will end the inner for-loop with 4 pairs of 4-sized sorted subarrays.
    - The math.min() is there in case the end of the array doesn't have a subarray of that certain size (e.g., odd arrays might not have enough elements in the last subarray so math.min just takes N-1 as the hi-index instead).
- Once both the for loops are completed, the array will be sorted.

        - Note that the bottom-up version is about 10% slower than the recursive, top-down version.
- Sorting Complexity
    - Computational complexity
        - A framework to study the efficiency of algorithms for solving a particular problem X.
        - Model of computation
            - The operations that the algorithms are allowed to perform.
        - Cost model
            - Operation count(s).
                - Count the comparisons.
        - Upper bound
            - Cost guarantee provided by some algorithm for X.
        - Lower bound
            - Proven limit on cost guarantee of all algorithms for X.
        - Optimal algorithm
            - Algorithm with the best possible cost guarantee for X.
            - An algorithm where we prove that the upper bound and lower bound are the same.
- Comparators
    - Comparators are a Java mechanism that helps us sort the same data on different sort keys or different orders.
        - Think about ordering a music playlist by different keys.
            - You might want to order it by song name or artist name, maybe even the date released or length of the song.
    - So how do we arrange to do this type of multi-sorting on the same data in our Java sorts?
        - Previously, we looked at implementing Java's comparable interface to be able to sort any type of data.
        - Now we'll look at a different interface called the Comparator interface.
    - Comparator interface
        - Allows you to sort the same data using an alternate order.

```
public interface Comparator<Key>
-------------------------------------------------------------
              int  compare(Key v, Key w)      compare keys v and w
```

        -
        - Requires a **<u>total order</u>** to work (refer to previous notes above).

Ex. Sort strings by:

- Natural order. Now is the time
- Case insensitive. is Now the time
- Spanish. café cafetero cuarto churro nube ñoño
- British phone book. McKinley Mackintosh
- . . .

pre-1994 order for digraphs ch and ll and rr

        -

- Using comparators with Java's system sort
  - Java's in-built sort.
    - To use with Java system sort:
      - Create `Comparator` object.
      - Pass as second argument to `Arrays.sort()`.

```
String[] a;
...
Arrays.sort(a);
...
Arrays.sort(a, String.CASE_INSENSITIVE_ORDER);
...
Arrays.sort(a, Collator.getInstance(new Locale("es")));
...
Arrays.sort(a, new BritishPhoneBookOrder());
...
```

uses natural order

uses alternate order defined by Comparator<String> object

Bottom line. Decouples the definition of the data type from the definition of what it means to compare two objects of that type.

- Using comparators with our sorting methods
  - To support comparators in our sort implementations:
    - Use `Object` instead of `Comparable`.
    - Pass `Comparator` to `sort()` and `less()` and use it in `less()`.
  - Example

insertion sort using a Comparator

```
public static void sort(Object[] a, Comparator comparator)
{
   int N = a.length;
   for (int i = 0; i < N; i++)
      for (int j = i; j > 0 && less(comparator, a[j], a[j-1]); j--)
         exch(a, j, j-1);
}

private static boolean less(Comparator c, Object v, Object w)
{  return c.compare(v, w) < 0;   }

private static void exch(Object[] a, int i, int j)
{  Object swap = a[i]; a[i] = a[j]; a[j] = swap;  }
```

- Implementing a comparator
  - To implement a comparator:
    - Define a (nested) class that implements the `Comparator` interface.
    - Implement the `compare()` method.
  - Example for ordering a dataset of students.

```
public class Student
{
   public static final Comparator<Student> BY_NAME    = new ByName();
   public static final Comparator<Student> BY_SECTION = new BySection();
   private final String name;
   private final int section;
   ...

   private static class ByName implements Comparator<Student>
   {
      public int compare(Student v, Student w)
      {  return v.name.compareTo(w.name);  }
   }

   private static class BySection implements Comparator<Student>
   {
      public int compare(Student v, Student w)
      {  return v.section - w.section;  }
   }
}
```

one Comparator for the class

this technique works here since no danger of overflow

- As you can see, you have ByName and BySection as nested classes that implement Comparators for the Student object which have compare() methods.
- Depending on which order you want, a different compare method gets called since both ordering methods implement compare().

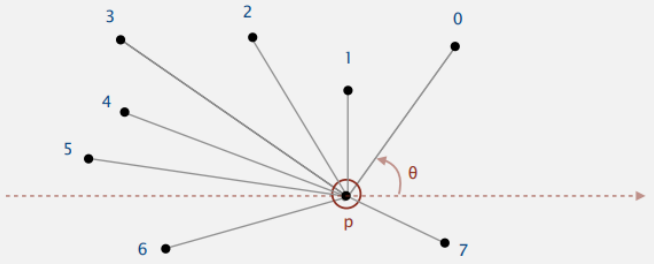
```
Arrays.sort(a, Student.BY_NAME);
```

Andrews	3	A	664-480-0023	097 Little
Battle	4	C	874-088-1212	121 Whitman
Chen	3	A	991-878-4944	308 Blair
Fox	3	A	884-232-5341	11 Dickinson
Furia	1	A	766-093-9873	101 Brown
Gazsi	4	B	766-093-9873	101 Brown
Kanaga	3	B	898-122-9643	22 Brown
Rohde	2	A	232-343-5555	343 Forbes

```
Arrays.sort(a, Student.BY_SECTION);
```

Furia	1	A	766-093-9873	101 Brown
Rohde	2	A	232-343-5555	343 Forbes
Andrews	3	A	664-480-0023	097 Little
Chen	3	A	991-878-4944	308 Blair
Fox	3	A	884-232-5341	11 Dickinson
Kanaga	3	B	898-122-9643	22 Brown
Battle	4	C	874-088-1212	121 Whitman
Gazsi	4	B	766-093-9873	101 Brown

- Example
  - Polar order
    - Given a point p, order points by the polar angle they make with p.




```
Arrays.sort(points, p.POLAR_ORDER);
```

Application. Graham scan algorithm for convex hull. [see previous lecture]

High-school trig solution. Compute polar angle θ w.r.t. p using `atan2()`.

Drawback. Evaluating a trigonometric function is expensive.

A ccw-based solution.

- If q_1 is above p and q_2 is below p, then q_1 makes smaller polar angle.
- If q_1 is below p and q_2 is above p, then q_1 makes larger polar angle.
- Otherwise, $ccw(p, q_1, q_2)$ identifies which of q_1 or q_2 makes larger angle.

- Implementation

```
public class Point2D
{
   public final Comparator<Point2D> POLAR_ORDER = new PolarOrder();
   private final double x, y;
   ...

   private static int ccw(Point2D a, Point2D b, Point2D c)
   {  /* as in previous lecture */  }

   private class PolarOrder implements Comparator<Point2D>
   {
      public int compare(Point2D q1, Point2D q2)
      {
         double dy1 = q1.y - y;
         double dy2 = q2.y - y;

         if      (dy1 == 0 && dy2 == 0) {  ...  }
         else if (dy1 >= 0 && dy2 < 0) return -1;
         else if (dy2 >= 0 && dy1 < 0) return +1;
         else return -ccw(Point2D.this, q1, q2);
      }
   }
}
```

one Comparator for each point (not static)

← p, q1, q2 horizontal
← q1 above p; q2 below p
← q1 below p; q2 above p
← both above or below p

to access invoking point from within inner class

- Stability
  - The concept that already sorted elements will be moved around when ordered by a different order (unstable if they do, stable if they don't).
  - There are stable sorts such as insertion sort and mergesort and unstable sorts such as selection sort, shell sort, and quicksort.
  - Let's say you were to order a list of students by first name, and then order them by section.
    - A stable sort would order the list by section, but for students with the same section, it would keep them in order by first name since we originally ordered it by first name. So the list would be primarily sorted by section, then for multiple students that are in the same section, they would stay sorted by first name.
    - An unstable sort would order the list by first name, then when you run the section order, it would completely order the list by section, and students in the same section would just be jumbled.
- Collinear Points assignment grade: 100%