2.2 Mergesort

- mergesort
- bottom-up mergesort
- sorting complexity
- comparators
- stability
Two classic sorting algorithms

Critical components in the world’s computational infrastructure.
- Full scientific understanding of their properties has enabled us to develop them into practical system sorts.
- Quicksort honored as one of top 10 algorithms of 20\textsuperscript{th} century in science and engineering.

Mergesort. [this lecture]
- Java sort for objects.
- Perl, C++ stable sort, Python stable sort, Firefox JavaScript, ...

Quicksort. [next lecture]
- Java sort for primitive types.
- C qsort, Unix, Visual C++, Python, Matlab, Chrome JavaScript, ...
- mergesort
- bottom-up mergesort
- sorting complexity
- comparators
- stability
Mergesort

Basic plan.
• Divide array into two halves.
• **Recursively** sort each half.
• Merge two halves.

```
input  M E R G E S O R T E X A M P L E
sort left half  E E G M O R R S    T E X A M P L E
sort right half E E G M O R R S    A E E L M P T X
merge results  A E E E E E G L M M O P R R S T X
```

Mergesort overview

*First Draft of a Report on the EDVAC*

John von Neumann
Merging demo
**Merging**

**Q.** How to combine two sorted subarrays into a sorted whole.

**A.** Use an auxiliary array.

```
<table>
<thead>
<tr>
<th>k</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>input</td>
<td>E</td>
<td>E</td>
<td>G</td>
<td>M</td>
<td>R</td>
<td>A</td>
<td>C</td>
<td>E</td>
<td>R</td>
<td>T</td>
</tr>
<tr>
<td>copy</td>
<td>E</td>
<td>E</td>
<td>G</td>
<td>M</td>
<td>R</td>
<td>A</td>
<td>C</td>
<td>E</td>
<td>R</td>
<td>T</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>i</th>
<th>j</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>input</td>
<td></td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>copy</td>
<td>E</td>
<td>E</td>
<td>G</td>
<td>M</td>
<td>R</td>
<td>A</td>
<td>C</td>
<td>E</td>
<td>R</td>
<td>T</td>
<td></td>
</tr>
</tbody>
</table>

Abstract in-place merge trace
```

merged result

<table>
<thead>
<tr>
<th>A</th>
<th>C</th>
<th>E</th>
<th>E</th>
<th>E</th>
<th>G</th>
<th>M</th>
<th>R</th>
<th>R</th>
<th>T</th>
</tr>
</thead>
</table>

```
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++)
        aux[k] = a[k];

    int i = lo, j = mid+1;
    for (int k = lo; k <= hi; k++)
    {
        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
}
Assertions

**Assertion.** Statement to test assumptions about your program.
- Helps detect logic bugs.
- Documents code.

**Java assert statement.** Throws an exception unless boolean condition is true.

```java
assert isSorted(a, lo, hi);
```

**Can enable or disable at runtime.** ⇒ No cost in production code.

```java
java -ea MyProgram   // enable assertions
java -da MyProgram   // disable assertions (default)
```

**Best practices.** Use to check internal invariants. Assume assertions will be disabled in production code (so do not use for external argument-checking).
public class Merge
{
    private static void merge(Comparable[] a, Comparable[] aux, int lo, int mid, int hi)
    { /* 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);
    }
}
Mergesort: trace

Trace of merge results for top-down mergesort

<table>
<thead>
<tr>
<th>lo</th>
<th>hi</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>12</td>
<td>13</td>
</tr>
<tr>
<td>14</td>
<td>15</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>a[]</th>
</tr>
</thead>
<tbody>
<tr>
<td>M E R G E S O R T E X A M P L E</td>
</tr>
<tr>
<td>E M R G E S O R T E X A M P L E</td>
</tr>
<tr>
<td>E M G R E S O R T E X A M P L E</td>
</tr>
<tr>
<td>E G M R E S O R T E X A M P L E</td>
</tr>
<tr>
<td>E G M R E S O R T E X A M P L E</td>
</tr>
<tr>
<td>E G M R E S O R T E X A M P L E</td>
</tr>
<tr>
<td>E G M R E O R S T E X A M P L E</td>
</tr>
<tr>
<td>E E G M O R R S T E X A M P L E</td>
</tr>
<tr>
<td>E E G M O R R S T E X A M P L E</td>
</tr>
<tr>
<td>E E G M O R R S T E X A M P L E</td>
</tr>
<tr>
<td>E E G M O R R S T E X A M P L E</td>
</tr>
<tr>
<td>E E G M O R R S T E X A M P L E</td>
</tr>
<tr>
<td>E E G M O R R S T E X A M P L E</td>
</tr>
<tr>
<td>E E G M O R R S T E X A M P L E</td>
</tr>
<tr>
<td>E E G M O R R S T E X A M P L E</td>
</tr>
<tr>
<td>E E G M O R R S T E X A M P L E</td>
</tr>
</tbody>
</table>

result after recursive call
Mergesort: animation

50 random items

http://www.sorting-algorithms.com/merge-sort
Mergesort: animation

50 reverse-sorted items

http://www.sorting-algorithms.com/merge-sort
Mergesort: Transylvanian-Saxon folk dance
Mergesort: empirical analysis

Running time estimates:
- Laptop executes $10^8$ compares/second.
- Supercomputer executes $10^{12}$ compares/second.

<table>
<thead>
<tr>
<th></th>
<th>insertion sort ($N^2$)</th>
<th>mergesort ($N \log N$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>computer</td>
<td>thousand</td>
<td>million</td>
</tr>
<tr>
<td>home</td>
<td>instant</td>
<td>2.8 hours</td>
</tr>
<tr>
<td>super</td>
<td>instant</td>
<td>1 second</td>
</tr>
</tbody>
</table>
Proposition. Mergesort uses at most $N \lg N$ compares and $6N \lg N$ array accesses to sort any array of size $N$.

Pf sketch. The number of compares $C(N)$ and array accesses $A(N)$ to mergesort an array of size $N$ satisfy the recurrences:

\[
C(N) \leq C([N/2]) + C([N/2]) + N \quad \text{for } N > 1, \text{ with } C(1) = 0.
\]

\[
A(N) \leq A([N/2]) + A([N/2]) + 6N \quad \text{for } N > 1, \text{ with } A(1) = 0.
\]

We solve the recurrence when $N$ is a power of 2. (see COS 340)

\[
D(N) = 2D(N/2) + N, \text{ for } N > 1, \text{ with } D(1) = 0.
\]
Proposition. If $D(N)$ satisfies $D(N) = 2D(N/2) + N$ for $N > 1$, with $D(1) = 0$, then $D(N) = N \lg N$.

Pf 1. [assuming $N$ is a power of 2]

Divide-and-conquer recurrence: proof by picture

- $D(N) = N$
- $2(N/2) = N$
- $4(N/4) = N$
- $2^k(N/2^k) = N$
- $N/2(2) = N$

$N \lg N$
Divide-and-conquer recurrence: proof by expansion

**Proposition.** If $D(N)$ satisfies $D(N) = 2D(N/2) + N$ for $N > 1$, with $D(1) = 0$, then $D(N) = N\log N$.

**Pf 2.** [assuming $N$ is a power of 2]

\[
D(N) = 2D(N/2) + N
\]
\[
\frac{D(N)}{N} = 2\frac{D(N/2)}{N} + 1
\]
\[
= D(N/2) / (N/2) + 1
\]
\[
= D(N/4) / (N/4) + 1 + 1
\]
\[
= D(N/8) / (N/8) + 1 + 1 + 1
\]
\[
\vdots
\]
\[
= D(N/N) / (N/N) + 1 + 1 + \ldots + 1
\]
\[
= \log N
\]

---

given

divide both sides by $N$

algebra

apply to first term

apply to first term again

stop applying, $D(1) = 0$
Proposition. If $D(N)$ satisfies $D(N) = 2D(N/2) + N$ for $N > 1$, with $D(1) = 0$, then $D(N) = N\lg N$.

Pf 3. [assuming $N$ is a power of 2]
- Base case: $N = 1$.
- Inductive hypothesis: $D(N) = N\lg N$.
- Goal: show that $D(2N) = (2N)\lg (2N)$.

$$
D(2N) = 2D(N) + 2N \\
= 2N\lg N + 2N \\
= 2N(\lg (2N) - 1) + 2N \\
= 2N\lg (2N)
$$
**Mergesort analysis: memory**

**Proposition.** Mergesort uses extra space proportional to $N$.

**Pf.** The array aux[] needs to be of size $N$ for the last merge.

**Def.** A sorting algorithm is **in-place** if it uses $\leq c \log N$ extra memory.

**Ex.** Insertion sort, selection sort, shellsort.

**Challenge for the bored.** In-place merge. [Kronrod, 1969]
Mergesort: practical improvements

Use insertion sort for small subarrays.

- Mergesort has too much overhead for tiny subarrays.
- Cutoff to insertion sort for \( \approx 7 \) items.

```java
private static void sort(Comparable[] a, Comparable[] aux, int lo, int hi)
{
   if (hi <= lo + CUTOFF - 1) Insertion.sort(a, lo, hi);
   int mid = lo + (hi - lo) / 2;
   sort (a, aux, lo, mid);
   sort (a, aux, mid+1, hi);
   merge(a, aux, lo, mid, hi);
}
```
Mergesort: practical improvements

Stop if already sorted.

- Is biggest item in first half ≤ smallest item in second half?
- Helps for partially-ordered arrays.

```java
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);
    if (!less(a[mid+1], a[mid])) return;
    merge(a, aux, lo, mid, hi);
}
```
Mergesort: practical improvements

Eliminate the copy to the auxiliary array. Save time (but not space) by switching the role of the input and auxiliary array in each recursive call.

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

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

switch roles of aux[] and a[]
Mergesort: visualization

Visual trace of top-down mergesort for with cutoff for small subarrays

- first subarray
- second subarray
- first merge
- first half sorted
- second half sorted
- result
• mergesort
• bottom-up mergesort
• sorting complexity
• comparators
• stability
**Bottom-up mergesort**

**Basic plan.**

- Pass through array, merging subarrays of size 1.
- Repeat for subarrays of size 2, 4, 8, 16, ....

<table>
<thead>
<tr>
<th>sz = 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>merge(a, 0, 0, 1)</td>
</tr>
<tr>
<td>merge(a, 2, 2, 3)</td>
</tr>
<tr>
<td>merge(a, 4, 4, 5)</td>
</tr>
<tr>
<td>merge(a, 6, 6, 7)</td>
</tr>
<tr>
<td>merge(a, 8, 8, 9)</td>
</tr>
<tr>
<td>merge(a, 10, 10, 11)</td>
</tr>
<tr>
<td>merge(a, 12, 12, 13)</td>
</tr>
<tr>
<td>merge(a, 14, 14, 15)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>sz = 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>merge(a, 0, 1, 3)</td>
</tr>
<tr>
<td>merge(a, 4, 5, 7)</td>
</tr>
<tr>
<td>merge(a, 8, 9, 11)</td>
</tr>
<tr>
<td>merge(a, 12, 13, 15)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>sz = 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>merge(a, 0, 3, 7)</td>
</tr>
<tr>
<td>merge(a, 8, 11, 15)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>sz = 8</th>
</tr>
</thead>
<tbody>
<tr>
<td>merge(a, 0, 7, 15)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>a[i]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>M E R G E S O R T E X A M P L E</td>
</tr>
<tr>
<td>E M R G E S O R T E X A M P L E</td>
</tr>
<tr>
<td>E M R G E S O R T E X A M P L E</td>
</tr>
<tr>
<td>E M R G E S O R T E X A M P L E</td>
</tr>
<tr>
<td>E M R G E S O R T E X A M P L E</td>
</tr>
<tr>
<td>E M R G E S O R T E X A M P L E</td>
</tr>
<tr>
<td>E M R G E S O R T E X A M P L E</td>
</tr>
<tr>
<td>E M R G E S O R T E X A M P L E</td>
</tr>
<tr>
<td>E M R G E S O R T E X A M P L E</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>sz = 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>merge(a, 0, 1, 3)</td>
</tr>
<tr>
<td>merge(a, 4, 5, 7)</td>
</tr>
<tr>
<td>merge(a, 8, 9, 11)</td>
</tr>
<tr>
<td>merge(a, 12, 13, 15)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>sz = 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>merge(a, 0, 3, 7)</td>
</tr>
<tr>
<td>merge(a, 8, 11, 15)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>sz = 8</th>
</tr>
</thead>
<tbody>
<tr>
<td>merge(a, 0, 7, 15)</td>
</tr>
</tbody>
</table>

**Bottom line.** No recursion needed!
Bottom-up mergesort: Java implementation

```java
public class MergeBU {
    private static Comparable[] aux;

    private static void merge(Comparable[] a, int lo, int mid, int hi) {
        /* as before */
    }

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

Bottom line. Concise industrial-strength code, if you have the space.
Bottom-up mergesort: visual trace
- mergesort
- bottom-up mergesort
- sorting complexity
- comparators
- stability
Complexity of sorting

Computational complexity. Framework to study efficiency of algorithms for solving a particular problem \( X \).

Model of computation. Allowable operations.
Cost model. Operation count(s).
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 best possible cost guarantee for \( X \).

Example: sorting.

- Model of computation: decision tree.
- Cost model: \# compares.
- Upper bound: \( \sim N \lg N \) from mergesort.
- Lower bound: ?
- Optimal algorithm: ?
Decision tree (for 3 distinct items a, b, and c)

- a < b
  - b < c
    - yes
      - a b c
    - no
      - a < c
        - yes
          - a c b
        - no
          - c a b
  - no
    - a < c
      - yes
        - b c a
      - no
        - b < c
          - yes
            - c b a
          - no
            - (at least) one leaf for each possible ordering

code between compares (e.g., sequence of exchanges)

height of tree = worst-case number of compares
Proposition. Any compare-based sorting algorithm must use at least $\lg (N!) \sim N \lg N$ compares in the worst-case.

Pf.

- Assume array consists of $N$ distinct values $a_1$ through $a_N$.
- Worst case dictated by height $h$ of decision tree.
- Binary tree of height $h$ has at most $2^h$ leaves.
- $N!$ different orderings $\Rightarrow$ at least $N!$ leaves.
Compare-based lower bound for sorting

**Proposition.** Any compare-based sorting algorithm must use at least \( \lg (N!) \sim N \lg N \) compares in the worst-case.

**Pf.**
- Assume array consists of \( N \) distinct values \( a_1 \) through \( a_N \).
- Worst case dictated by height \( h \) of decision tree.
- Binary tree of height \( h \) has at most \( 2^h \) leaves.
- \( N! \) different orderings \( \Rightarrow \) at least \( N! \) leaves.

\[
2^h \geq \# \text{leaves} \geq N!
\Rightarrow h \geq \lg (N!) \sim N \lg N
\]

Stirling’s formula
Complexity of sorting

Model of computation. Allowable operations.

Cost model. Operation count(s).

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 best possible cost guarantee for $X$.

Example: sorting.

• Model of computation: decision tree.
• Cost model: # compares.
• Upper bound: $\sim N \log N$ from mergesort.
• Lower bound: $\sim N \log N$.
• Optimal algorithm = mergesort.

First goal of algorithm design: optimal algorithms.
Complexity results in context

Other operations? Mergesort is optimal with respect to number of compares (e.g., but not with respect to number of array accesses).

Space?
• Mergesort is not optimal with respect to space usage.
• Insertion sort, selection sort, and shellsort are space-optimal.

Challenge. Find an algorithm that is both time- and space-optimal. [stay tuned]

Lessons. Use theory as a guide.
Ex. Don't try to design sorting algorithm that guarantees \( \frac{1}{2} N \lg N \) compares.
Complexity results in context (continued)

Lower bound may not hold if the algorithm has information about:
• The initial order of the input.
• The distribution of key values.
• The representation of the keys.

Partially-ordered arrays. Depending on the initial order of the input, we may not need $N \lg N$ compares.

Duplicate keys. Depending on the input distribution of duplicates, we may not need $N \lg N$ compares.

Digital properties of keys. We can use digit/character compares instead of key compares for numbers and strings.
› mergesort
› bottom-up mergesort
› sorting complexity
› comparators
› stability
Comparable interface: review

Comparable interface: sort using a type's **natural order**.

```java
public class Date implements Comparable<Date>
{
    private final int month, day, year;

    public Date(int m, int d, int y)
    {
        month = m;
        day   = d;
        year  = y;
    }

    public int compareTo(Date that)
    {
        if (this.year < that.year ) return -1;
        if (this.year > that.year ) return +1;
        if (this.month < that.month) return -1;
        if (this.month > that.month) return +1;
        if (this.day   < that.day  ) return -1;
        if (this.day   > that.day  ) return +1;
        return 0;
    }
}
```
Comparator interface

Comparator interface: sort using an alternate order.

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

Required property. Must be a total order.

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
- ...
**Comparator interface: system sort**

To use with Java system sort:

- Create **Comparator** object.
- Pass as second argument to `Arrays.sort()`.

**Bottom line.** Decouples the definition of the data type from the definition of what it means to compare two objects of that type.
Comparator interface: using with our sorting libraries

To support comparators in our sort implementations:

- **Use** `Object` **instead of** `Comparable`.
- **Pass** `Comparator` **to** `sort()` **and** `less()` **and use it in** `less()`.

insertion sort using a Comparator

```java
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;  }
```
Comparator interface: implementing

To implement a comparator:

- Define a (nested) class that implements the Comparator interface.
- Implement the compare() method.

```java
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;
        }
    }
}
```

This technique works here since no danger of overflow.
Comparator interface: implementing

To implement a comparator:

- Define a (nested) class that implements the Comparator interface.
- Implement the compare() method.

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

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

<table>
<thead>
<tr>
<th>Name</th>
<th>ID</th>
<th>Section</th>
<th>Phone</th>
<th>Room</th>
</tr>
</thead>
<tbody>
<tr>
<td>Andrews</td>
<td>3</td>
<td>A</td>
<td>664-480-0023</td>
<td>097 Little</td>
</tr>
<tr>
<td>Battle</td>
<td>4</td>
<td>C</td>
<td>874-088-1212</td>
<td>121 Whitman</td>
</tr>
<tr>
<td>Chen</td>
<td>3</td>
<td>A</td>
<td>991-878-4944</td>
<td>308 Blair</td>
</tr>
<tr>
<td>Fox</td>
<td>3</td>
<td>A</td>
<td>884-232-5341</td>
<td>11 Dickinson</td>
</tr>
<tr>
<td>Furia</td>
<td>1</td>
<td>A</td>
<td>766-093-9873</td>
<td>101 Brown</td>
</tr>
<tr>
<td>Gazsi</td>
<td>4</td>
<td>B</td>
<td>766-093-9873</td>
<td>101 Brown</td>
</tr>
<tr>
<td>Kanaga</td>
<td>3</td>
<td>B</td>
<td>898-122-9643</td>
<td>22 Brown</td>
</tr>
<tr>
<td>Rohde</td>
<td>2</td>
<td>A</td>
<td>232-343-5555</td>
<td>343 Forbes</td>
</tr>
<tr>
<td>Furia</td>
<td>1</td>
<td>A</td>
<td>766-093-9873</td>
<td>101 Brown</td>
</tr>
<tr>
<td>Rohde</td>
<td>2</td>
<td>A</td>
<td>232-343-5555</td>
<td>343 Forbes</td>
</tr>
<tr>
<td>Andrews</td>
<td>3</td>
<td>A</td>
<td>664-480-0023</td>
<td>097 Little</td>
</tr>
<tr>
<td>Chen</td>
<td>3</td>
<td>A</td>
<td>991-878-4944</td>
<td>308 Blair</td>
</tr>
<tr>
<td>Fox</td>
<td>3</td>
<td>A</td>
<td>884-232-5341</td>
<td>11 Dickinson</td>
</tr>
<tr>
<td>Kanaga</td>
<td>3</td>
<td>B</td>
<td>898-122-9643</td>
<td>22 Brown</td>
</tr>
<tr>
<td>Battle</td>
<td>4</td>
<td>C</td>
<td>874-088-1212</td>
<td>121 Whitman</td>
</tr>
<tr>
<td>Gazsi</td>
<td>4</td>
<td>B</td>
<td>766-093-9873</td>
<td>101 Brown</td>
</tr>
</tbody>
</table>
Polar order

Polar order. Given a point \( p \), order points by the polar angle they make with \( p \).

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

High-school trig solution. Compute polar angle \( \theta \) w.r.t. \( p \) using \( \text{atan2()} \).

Drawback. Evaluating a trigonometric function is expensive.
Polar order

Given a point $p$, order points by the polar angle they make with $p$.

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 polar angle.

Arrays.sort(points, p.POLAR_ORDER);
Comparator interface: polar order

```java
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 dx1 = q1.x - x;
            double dy1 = q1.y - y;

            if       (dy1 == 0 && dy2 == 0) { ... } // p, q1, q2 horizontal
            else if (dy1 >= 0 && dy2 < 0) return -1; // q1 above p; q2 below p
            else if (dy2 >= 0 && dy1 < 0) return +1; // q1 below p; q2 above p
            else return -ccw(Point2D.this, q1, q2); // both above or below p
        }
    }
}
```

- One Comparator for each point (not static)
- To access invoking point from within inner class

- mergesort
- bottom-up mergesort
- sorting complexity
- comparators
- stability
Stability

A typical application. First, sort by name; then sort by section.

Selection.sort(a, Student.BY_NAME);

Selection.sort(a, Student.BY_SECTION);

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

@##%&@! Students in section 3 no longer sorted by name.

A stable sort preserves the relative order of items with equal keys.
Stability

Q. Which sorts are stable?
A. Insertion sort and mergesort (but not selection sort or shellsort).

Note. Need to carefully check code ("less than" vs "less than or equal to").
Stability: insertion sort

Proposition. Insertion sort is stable.

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

Pf. Equal items never move past each other.
Stability: selection sort

**Proposition.** Selection sort is not stable.

```java
public class Selection {
    public static void sort(Comparable[] a) {
        int N = a.length;
        for (int i = 0; i < N; i++) {
            int min = i;
            for (int j = i+1; j < N; j++)
                if (less(a[j], a[min]))
                    min = j;
            exch(a, i, min);
        }
    }
}
```

**Pf by counterexample.** Long-distance exchange might move an item past some equal item.  

<table>
<thead>
<tr>
<th>i</th>
<th>min</th>
<th>0</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td>B₁</td>
<td>B₂</td>
<td>A</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>A</td>
<td>B₂</td>
<td>B₁</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>A</td>
<td>B₂</td>
<td>B₁</td>
</tr>
</tbody>
</table>

```
Stability: shellsort

Proposition. Shellsort sort is not stable.

Pf by counterexample. Long-distance exchanges.
Stability: mergesort

Proposition. Mergesort is stable.

Pf. Suffices to verify that merge operation is stable.
Proposition. Merge operation is stable.

```
private static void merge(Comparable[] a, int lo, int mid, int hi)
{
    for (int k = lo; k <= hi; k++)
       aux[k] = a[k];

    int i = lo, j = mid+1;
    for (int k = lo; k <= hi; k++)
    {
        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++];
    }
}
```

Pf. Takes from left subarray if equal keys.