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 20th 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.

Mergesort overview

First Draft of a Report on the EDVAC

John von Neumann
Merging demo
**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>
<tr>
<td>i</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>j</td>
<td>0</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **Abstract in-place merge trace**

```
input
E E G M R A C E R T
copy
E E G M R A C E R T
k 0 1 2 3 4 5 6 7 8 9 i 0 1 2 3 4 5 6 7 8 9 j 0 5
0 A 0 6 E E G M R A C E R T
1 A C 0 7 E E G M R C E R T
2 A C E 1 7 E E G M R E R T
3 A C E E 2 7 E G M R E R T
4 A C E E E 2 8 G M R E R T
5 A C E E E G 3 8 G M R R T
6 A C E E E G M 4 8 M R R T
7 A C E E E G M R 5 8 R R T
8 A C E E E G M R R 5 9 R R T
9 A C E E E G M R R T 6 10 R T
merged result
A C E E E G M R R T
```
Merging: Java implementation

```java
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 (e.g., don't 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

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: empirical analysis

Running time estimates:

- Laptop executes $10^8$ compares/second.
- Supercomputer executes $10^{12}$ compares/second.

Bottom line. Good algorithms are better than supercomputers.
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:

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

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

We solve the simpler divide-and-conquer recurrence when $N$ is a power of 2.

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

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 2. [assuming \( N \) is a power of 2]

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

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 \( D(2N) = (2N) \lg (2N) \).

\[
D(2N) = 2D(N) + 2N \\
= 2N \lg N + 2N \\
= 2N (\lg (2N) - 1) + 2N \\
= 2N \lg (2N)
\]

given

inductive hypothesis

algebra

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

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

---

**Def.** A sorting algorithm is **in-place** if it uses $O(\log N)$ 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 \(\leq\) smallest item in second half?
• Helps for partially-ordered arrays.

```java
class Mergesort
{
    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[]
merge from a[] to aux[]
Mergesort: visualization

- first subarray
- second subarray
- first merge
- first half sorted
- second half sorted
- result

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

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
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: $\sim$?
- Optimal algorithm: $\sim$?
Decision tree (for 3 distinct items a, b, and c)

- **a < b**
  - yes
    - b < c
      - yes
        - a b c
      - no
        - a c b
  - no
    - a < c
      - yes
        - b a c
      - no
        - c a b

- **b < c**
  - yes
    - b < c
      - yes
        - b c a
      - no
        - c b a
  - no
    - a < c
      - yes
        - a c b
      - no
        - c a b

**height of tree = worst-case number of compares**

(at least) one leaf for each possible ordering
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.
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: $\sim N \lg N$ from mergesort.
- Lower bound: $\sim N \lg N$.
- 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.
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
Sort music library by artist name

<table>
<thead>
<tr>
<th>Name</th>
<th>Artist</th>
<th>Time</th>
<th>Album</th>
</tr>
</thead>
<tbody>
<tr>
<td>Let It Be</td>
<td>The Beatles</td>
<td>4:03</td>
<td>Let It Be</td>
</tr>
<tr>
<td>Take My Breath Away</td>
<td>BERLIN</td>
<td>4:13</td>
<td>Top Gun - Soundtrack</td>
</tr>
<tr>
<td>Circle Of Friends</td>
<td>Better Than Ezra</td>
<td>3:27</td>
<td>Empire Records</td>
</tr>
<tr>
<td>Dancing With Myself</td>
<td>Billy Idol</td>
<td>4:43</td>
<td>Don't Stop</td>
</tr>
<tr>
<td>Rebel Yell</td>
<td>Billy Idol</td>
<td>4:49</td>
<td>Rebel Yell</td>
</tr>
<tr>
<td>Piano Man</td>
<td>Billy Joel</td>
<td>5:36</td>
<td>Greatest Hits Vol. 1</td>
</tr>
<tr>
<td>Atomic</td>
<td>Blondie</td>
<td>3:50</td>
<td>Atomic: The Very Best Of Blondie</td>
</tr>
<tr>
<td>Sunday Girl</td>
<td>Blondie</td>
<td>3:15</td>
<td>Atomic: The Very Best Of Blondie</td>
</tr>
<tr>
<td>Call Me</td>
<td>Blondie</td>
<td>3:33</td>
<td>Atomic: The Very Best Of Blondie</td>
</tr>
<tr>
<td>Dreaming</td>
<td>Blondie</td>
<td>3:06</td>
<td>Atomic: The Very Best Of Blondie</td>
</tr>
<tr>
<td>Hurricane</td>
<td>Bob Dylan</td>
<td>8:32</td>
<td>Desire</td>
</tr>
<tr>
<td>The Times They Are A-Changin'</td>
<td>Bob Dylan</td>
<td>3:17</td>
<td>Greatest Hits</td>
</tr>
<tr>
<td>Livin' On A Prayer</td>
<td>Bon Jovi</td>
<td>4:11</td>
<td>Cross Road</td>
</tr>
<tr>
<td>Beds Of Roses</td>
<td>Bon Jovi</td>
<td>6:35</td>
<td>Cross Road</td>
</tr>
<tr>
<td>Runaway</td>
<td>Bon Jovi</td>
<td>3:53</td>
<td>Cross Road</td>
</tr>
<tr>
<td>Rasputin (Extended Mix)</td>
<td>Boney M</td>
<td>5:50</td>
<td>Greatest Hits</td>
</tr>
<tr>
<td>Have You Ever Seen The Rain</td>
<td>Bonnie Tyler</td>
<td>4:10</td>
<td>Faster Than The Speed Of Night</td>
</tr>
<tr>
<td>Total Eclipse Of The Heart</td>
<td>Bonnie Tyler</td>
<td>7:02</td>
<td>Faster Than The Speed Of Night</td>
</tr>
<tr>
<td>Straight From The Heart</td>
<td>Bonnie Tyler</td>
<td>3:41</td>
<td>Faster Than The Speed Of Night</td>
</tr>
<tr>
<td>Holding Out For A Hero</td>
<td>Bonny Tyler</td>
<td>5:49</td>
<td>Meat Loaf And Friends</td>
</tr>
<tr>
<td>Dancing In The Dark</td>
<td>Bruce Springsteen</td>
<td>4:05</td>
<td>Born In The U.S.A.</td>
</tr>
<tr>
<td>Thunder Road</td>
<td>Bruce Springsteen</td>
<td>4:51</td>
<td>Born To Run</td>
</tr>
<tr>
<td>Born To Run</td>
<td>Bruce Springsteen</td>
<td>4:30</td>
<td>Born To Run</td>
</tr>
<tr>
<td>Jungleland</td>
<td>Bruce Springsteen</td>
<td>9:34</td>
<td>Born To Run</td>
</tr>
</tbody>
</table>
Sort music library by song name

<table>
<thead>
<tr>
<th>Name</th>
<th>Artist</th>
<th>Time</th>
<th>Album</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alive</td>
<td>Pearl Jam</td>
<td>5:41</td>
<td>Ten</td>
</tr>
<tr>
<td>All Over The World</td>
<td>Pixies</td>
<td>5:27</td>
<td>Bossanova</td>
</tr>
<tr>
<td>All Through The Night</td>
<td>Cyndi Lauper</td>
<td>4:30</td>
<td>She's So Unusual</td>
</tr>
<tr>
<td>Allison Road</td>
<td>Gin Blossoms</td>
<td>3:19</td>
<td>New Miserable Experience</td>
</tr>
<tr>
<td>Amor, Amor, Amor Y Ensancha El</td>
<td>Extremoduro</td>
<td>2:34</td>
<td>Delta (1992)</td>
</tr>
<tr>
<td>And We Danced</td>
<td>Hooters</td>
<td>3:50</td>
<td>Nervous Night</td>
</tr>
<tr>
<td>As I Lay Me Down</td>
<td>Sophie B. Hawkins</td>
<td>4:09</td>
<td>Whaler</td>
</tr>
<tr>
<td>Atomic</td>
<td>Blondie</td>
<td>3:50</td>
<td>Atomic: The Very Best Of Blondie</td>
</tr>
<tr>
<td>Automatic Lover</td>
<td>Jay-Jay Johnson</td>
<td>4:19</td>
<td>Antenna</td>
</tr>
<tr>
<td>Baba O'Reilly</td>
<td>The Who</td>
<td>5:01</td>
<td>Who's Better, Who's Best</td>
</tr>
<tr>
<td>Beautiful Life</td>
<td>Ace Of Base</td>
<td>3:40</td>
<td>The Bridge</td>
</tr>
<tr>
<td>Beds Of Roses</td>
<td>Bon Jovi</td>
<td>6:35</td>
<td>Cross Road</td>
</tr>
<tr>
<td>Black</td>
<td>Pearl Jam</td>
<td>5:44</td>
<td>Ten</td>
</tr>
<tr>
<td>Bleed American</td>
<td>Jimmy Eat World</td>
<td>3:04</td>
<td>Bleed American</td>
</tr>
<tr>
<td>Borderline</td>
<td>Madonna</td>
<td>4:00</td>
<td>The Immaculate Collection</td>
</tr>
<tr>
<td>Born To Run</td>
<td>Bruce Springsteen</td>
<td>4:30</td>
<td>Born To Run</td>
</tr>
<tr>
<td>Both Sides Of The Story</td>
<td>Phil Collins</td>
<td>6:43</td>
<td>Both Sides</td>
</tr>
<tr>
<td>Bouncing Around The Room</td>
<td>Phish</td>
<td>4:09</td>
<td>A Live One (Disc 1)</td>
</tr>
<tr>
<td>Boys Don't Cry</td>
<td>The Cure</td>
<td>2:35</td>
<td>Staring At The Sea: The Singles 1979–1985</td>
</tr>
<tr>
<td>Brat</td>
<td>Green Day</td>
<td>1:43</td>
<td>Isnomiac</td>
</tr>
<tr>
<td>Breakdown</td>
<td>Deerheart</td>
<td>3:40</td>
<td>Deerheart</td>
</tr>
<tr>
<td>Bring Me To Life (Kevin Roen Mix)</td>
<td>Evanescence Vs. Pa...</td>
<td>9:48</td>
<td></td>
</tr>
<tr>
<td>Californication</td>
<td>Red Hot Chili Peppers</td>
<td>1:40</td>
<td></td>
</tr>
<tr>
<td>Call Me</td>
<td>Blondie</td>
<td>3:33</td>
<td>Atomic: The Very Best Of Blondie</td>
</tr>
<tr>
<td>Can't Get You Out Of My Head</td>
<td>Kylie Minogue</td>
<td>3:50</td>
<td>Fever</td>
</tr>
<tr>
<td>Celebration</td>
<td>Kool &amp; The Gang</td>
<td>3:45</td>
<td>Time Life Music Sounds Of The Seventies – C</td>
</tr>
<tr>
<td>Chance Changes</td>
<td>Sukhwinder Singh</td>
<td>5:11</td>
<td>Bombay Dreams</td>
</tr>
</tbody>
</table>
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
- ...

pre-1994 order for digraphs ch and ll and rr
Comparator interface: system 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());
...

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);
   }
}
```

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

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

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

```java
public class Student {
    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);  // This trick can be used only when no danger of overflow.
        }
    }

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

    public static final Comparator<Student> BY_NAME    = new ByName();
    public static final Comparator<Student> BY_SECTION = new BySection();
}
```
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>Student</th>
<th>ID</th>
<th>Name</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>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Student</th>
<th>ID</th>
<th>Name</th>
<th>Phone</th>
<th>Room</th>
</tr>
</thead>
<tbody>
<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

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.
Comparator interface: polar order

```java
public class Point2D {
    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) { ... }
            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)

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

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

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

```javascript
Selection.sort(a, Student.BY_NAME);
```

<table>
<thead>
<tr>
<th>Name</th>
<th>Sect</th>
<th>Key</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>
</tbody>
</table>

```javascript
Selection.sort(a, Student.BY_SECTION);
```

<table>
<thead>
<tr>
<th>Name</th>
<th>Sect</th>
<th>Key</th>
<th>Phone</th>
<th>Room</th>
</tr>
</thead>
<tbody>
<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>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>Andrews</td>
<td>3</td>
<td>A</td>
<td>664-480-0023</td>
<td>097 Little</td>
</tr>
<tr>
<td>Kanaga</td>
<td>3</td>
<td>B</td>
<td>898-122-9643</td>
<td>22 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>Battle</td>
<td>4</td>
<td>C</td>
<td>874-088-1212</td>
<td>121 Whitman</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).

<table>
<thead>
<tr>
<th>sorted by time</th>
<th>sorted by location (not stable)</th>
<th>sorted by location (stable)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chicago 09:00:00</td>
<td>Chicago 09:25:52</td>
<td>Chicago 09:00:00</td>
</tr>
<tr>
<td>Phoenix 09:00:03</td>
<td>Chicago 09:03:13</td>
<td>Chicago 09:00:59</td>
</tr>
<tr>
<td>Houston 09:00:13</td>
<td>Chicago 09:21:05</td>
<td>Chicago 09:03:13</td>
</tr>
<tr>
<td>Chicago 09:00:59</td>
<td>Chicago 09:19:46</td>
<td>Chicago 09:19:32</td>
</tr>
<tr>
<td>Houston 09:01:10</td>
<td>Chicago 09:19:32</td>
<td>Chicago 09:19:46</td>
</tr>
<tr>
<td>Chicago 09:03:13</td>
<td>Chicago 09:00:00</td>
<td>Chicago 09:21:05</td>
</tr>
<tr>
<td>Seattle 09:10:11</td>
<td>Chicago 09:35:21</td>
<td>Chicago 09:25:52</td>
</tr>
<tr>
<td>Seattle 09:10:25</td>
<td>Chicago 09:00:59</td>
<td>Chicago 09:35:21</td>
</tr>
<tr>
<td>Phoenix 09:14:25</td>
<td>Houston 09:01:10</td>
<td>Houston 09:00:13</td>
</tr>
<tr>
<td>Chicago 09:19:32</td>
<td>Houston 09:00:13</td>
<td>Houston 09:01:10</td>
</tr>
<tr>
<td>Chicago 09:19:46</td>
<td>Phoenix 09:37:44</td>
<td>Phoenix 09:00:03</td>
</tr>
<tr>
<td>Chicago 09:21:05</td>
<td>Phoenix 09:00:03</td>
<td>Phoenix 09:14:25</td>
</tr>
<tr>
<td>Seattle 09:22:54</td>
<td>Seattle 09:10:25</td>
<td>Seattle 09:10:11</td>
</tr>
<tr>
<td>Chicago 09:25:52</td>
<td>Seattle 09:36:14</td>
<td>Seattle 09:10:25</td>
</tr>
<tr>
<td>Seattle 09:36:14</td>
<td>Seattle 09:10:11</td>
<td>Seattle 09:22:54</td>
</tr>
<tr>
<td>Phoenix 09:37:44</td>
<td>Seattle 09:22:54</td>
<td>Seattle 09:36:14</td>
</tr>
</tbody>
</table>

**Note.** Need to carefully check code ("less than" vs "less than or equal to").
Proposition. Insertion sort is stable.

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

Proposition. Selection sort is not stable.

Pf by counterexample. Long-distance exchange might move an item past some equal item.
Stability: shellsort

Proposition. Shell sort is not stable.

public class Shell
{
   public static void sort(Comparable[] a)
   {
      int N = a.length;
      int h = 1;
      while (h < N/3) h = 3*h + 1;
      while (h >= 1)
      {
         for (int i = h; i < N; i++)
         {
            for (int j = i; j > h && less(a[j], a[j-h]); j -= h)
               exch(a, j, j-h);
         }
         h = h/3;
      }
   }
}

Pf by counterexample. Long-distance exchanges.
Proposition. Mergesort is stable.

Pf. Suffices to verify that merge operation is stable.
Stability: mergesort

**Proposition.** Merge operation is stable.

```java
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.