Acknowledgement: The course slides are adapted from the slides prepared by R. Sedgewick and K. Wayne of Princeton University.

Mergesort

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

Abstract in-place merge

Goal. Given two sorted subarrays \(a[lo]\) to \(a[mid]\) and \(a[mid+1]\) to \(a[hi]\), replace with sorted subarray \(a[lo]\) to \(a[hi]\).
Abstract in-place merge

**Goal.** Given two sorted subarrays \(a[lo] \) to \(a[mid] \) and \(a[mid+1] \) to \(a[hi] \), replace with sorted subarray \(a[lo] \) to \(a[hi] \).

\[ a[lo] \quad E \quad E \quad G \quad M \quad R \quad A \quad C \quad E \quad R \quad T \]

\[ aux[i] \quad E \quad E \quad G \quad M \quad R \quad A \quad C \quad E \quad R \quad T \]

**Goal.** Given two sorted subarrays \(a[lo] \) to \(a[mid] \) and \(a[mid+1] \) to \(a[hi] \), replace with sorted subarray \(a[lo] \) to \(a[hi] \).

\[ a[lo] \quad A \quad E \quad G \quad M \quad R \quad A \quad C \quad E \quad R \quad T \]

\[ aux[i] \quad E \quad E \quad G \quad M \quad R \quad A \quad C \quad E \quad R \quad T \]

**Goal.** Given two sorted subarrays \(a[lo] \) to \(a[mid] \) and \(a[mid+1] \) to \(a[hi] \), replace with sorted subarray \(a[lo] \) to \(a[hi] \).

\[ aux[i] \quad E \quad E \quad G \quad M \quad R \quad A \quad C \quad E \quad R \quad T \]

**Goal.** Given two sorted subarrays \(a[lo] \) to \(a[mid] \) and \(a[mid+1] \) to \(a[hi] \), replace with sorted subarray \(a[lo] \) to \(a[hi] \).

\[ aux[i] \quad E \quad E \quad G \quad M \quad R \quad A \quad C \quad E \quad R \quad T \]
**Abstract in-place merge**

**Goal.** Given two sorted subarrays \(a[lo]\) to \(a[mid]\) and \(a[mid+1]\) to \(a[hi]\), replace with sorted subarray \(a[lo]\) to \(a[hi]\).

\[
\begin{array}{cccccccccc}
\text{aux[]} & \text{E} & \text{E} & \text{G} & \text{M} & \text{R} & \text{A} & \text{C} & \text{E} & \text{R} & \text{T} \\
i & j
\end{array}
\]
**Abstract in-place merge**

**Goal.** Given two sorted subarrays \(a[lo]\) to \(a[mid]\) and \(a[mid+1]\) to \(a[hi]\), replace with sorted subarray \(a[lo]\) to \(a[hi]\).

```plaintext
\[
\text{compare minimum in each subarray}
\]

aux[]: 

\[
\begin{array}{c}
E & E & G & M & R & A & C & E & R & T \\
\end{array}
\]

\[
i & j
\]
```
**Abstract in-place merge**

**Goal.** Given two sorted subarrays \( a[lo] \) to \( a[mid] \) and \( a[mid+1] \) to \( a[hi] \), replace with sorted subarray \( a[lo] \) to \( a[hi] \).

\[
\begin{array}{cccccccccc}
\text{a[\(k\)]} & A & C & E & E & E & G & C & E & R & T \\
\end{array}
\]

compare minimum in each subarray

\[
\begin{array}{cccccccccc}
\text{aux[i]} & E & E & G & M & R & A & C & E & R & T \\
\end{array}
\]

\[
\begin{array}{cccccccccc}
\text{a[\(k\)]} & A & C & E & E & E & G & C & E & R & T \\
\end{array}
\]

compare minimum in each subarray

\[
\begin{array}{cccccccccc}
\text{aux[i]} & E & E & G & M & R & A & C & E & R & T \\
\end{array}
\]
Goal. Given two sorted subarrays $a[lo]$ to $a[mid]$ and $a[mid+1]$ to $a[hi]$, replace with sorted subarray $a[lo]$ to $a[hi]$.

Abstract in-place merge

Consider subarrays $a[lo]$ to $a[mid]$ and $a[mid+1]$ to $a[hi]$.

1. Compare minimum in each subarray and choose the minimum.
2. Replace the subarrays with the chosen minimum.
3. If one subarray is exhausted, take from the other.

Example:

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

Next, compare minimum in each subarray and choose the minimum.

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

Next, if one subarray is exhausted, take from the other.

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

Next, compare minimum in each subarray and choose the minimum.

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

Next, if one subarray is exhausted, take from the other.

```
<table>
<thead>
<tr>
<th></th>
<th>E</th>
<th>E</th>
<th>G</th>
<th>M</th>
<th>R</th>
<th>A</th>
<th>C</th>
<th>E</th>
<th>R</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<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></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
Abstract in-place merge

Goal. Given two sorted subarrays $a[lo]$ to $a[mid]$ and $a[mid+1]$ to $a[hi]$, replace with sorted subarray $a[lo]$ to $a[hi]$.

one subarray exhausted, take from other

both subarrays exhausted, done

Merging

Q. How to combine two sorted subarrays into a sorted whole.
A. Use an auxiliary array.
Mergesort: Java implementation

```java
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

<table>
<thead>
<tr>
<th>lo</th>
<th>i</th>
<th>mid</th>
<th>j</th>
<th>hi</th>
</tr>
</thead>
<tbody>
<tr>
<td>a[]</td>
<td>A</td>
<td>G</td>
<td>L</td>
<td>O</td>
</tr>
</tbody>
</table>

Mergesort: animation

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.

<table>
<thead>
<tr>
<th>Insertion Sort ($N^2$)</th>
<th>Mergesort ($N \log N$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Computer</td>
<td>Thousand</td>
</tr>
<tr>
<td>home</td>
<td>instant</td>
</tr>
<tr>
<td>super</td>
<td>instant</td>
</tr>
</tbody>
</table>

Bottom line. Good algorithms are better than supercomputers.

Mergesort: number of compares and array accesses

**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:

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

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

We solve the recurrence when $N$ is a power of 2.

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

Divide-and-conquer recurrence: proof by picture

**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]

\[
\begin{align*}
D(N) &= N, \\
D(N/2) &= (N/2), \\
D(N/4) &= (N/4), \\
\vdots \\
D(1) &= 1.
\end{align*}
\]

\[
\begin{align*}
2 \times (N/2) &= N, \\
4 \times (N/4) &= N, \\
\vdots \\
N/2 &= N.
\end{align*}
\]

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

**Proof.**

1. **Assuming \( N \) is a power of 2**

   Divide-and-conquer recurrence: proof by expansion
   \[
   D(N) = 2D(N/2) + N
   \]
   given
   divide both sides by \( N \)
   algebra
   apply to first term
   apply to first term again
   stop applying, \( D(1) = 0 \)

2. **Base case:** \( N = 1 \).
3. **Inductive hypothesis:** \( D(N) = N \log N \).
4. **Goal:** show that \( D(2N) = (2N) \log (2N) \).

   Divide-and-conquer recurrence: proof by induction
   \[
   D(2N) = 2D(N) + 2N
   = 2N \log N + 2N
   = 2N (\log (2N) - 1) + 2N
   = 2N \log (2N)
   \]
   given
   inductive hypothesis
   algebra
   QED

**Mergesort analysis: memory**

**Proposition.** Mergesort uses extra space proportional to \( N \).

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

**Definition.** A sorting algorithm is **in-place** if it uses \( \leq c \log N \) extra memory.

**Example.** 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.

```
Mergesort: visualization
```

```
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: visualization
```

```
Mergesort: visualization
```

```
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++];
    }
}
```

```
Mergesort: visualization
```

```
Mergesort: visualization
```

```
Bottom-up mergesort
```

```
Bottom-up mergesort
```

Basic plan.

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

```
Bottom-up mergesort
```

```
Bottom-up mergesort
```

```
Bottom-up mergesort
```

```
Bottom line. No recursion needed!
```

```
Bottom-up mergesort
```

```
Bottom-up mergesort
```

```
Bottom-up mergesort
```

```
Bottom-up mergesort
```

```
Bottom-up mergesort
```

```
Bottom-up mergesort
```

```
Bottom-up mergesort
```

```
Bottom-up mergesort
```

```
Bottom-up mergesort
```

```
Bottom-up mergesort
```

```
Bottom-up mergesort
```

```
Bottom-up mergesort
```

```
Bottom-up mergesort
```

```
Bottom-up mergesort
```

```
Bottom-up mergesort
```

```
Bottom-up mergesort
```

```
Bottom-up mergesort
```

```
Bottom-up mergesort
```

```
Mergesort: visualization
```

```
Mergesort: visualization
```

```
Mergesort: visualization
```

```
Mergesort: visualization
```

```
Mergesort: visualization
```

```
Mergesort: visualization
```

```
Mergesort: visualization
```

```
Mergesort: visualization
```

```
Mergesort: visualization
```

```
Mergesort: visualization
```

```
Mergesort: visualization
```

```
Mergesort: visualization
```

```
Mergesort: visualization
```

```
Mergesort: visualization
```

```
Mergesort: visualization
```

```
Mergesort: visualization
```

```
Mergesort: visualization
```

```
Mergesort: visualization
```

```
Mergesort: visualization
```

```
Mergesort: visualization
```

```
Mergesort: visualization
```

```
Mergesort: visualization
```

```
Mergesort: visualization
```

```
Mergesort: visualization
```

```
Mergesort: visualization
```

```
Mergesort: visualization
```

```
Mergesort: visualization
```

```
Mergesort: visualization
```

```
Mergesort: visualization
```

```
Mergesort: visualization
```

```
Mergesort: visualization
```

```
Mergesort: visualization
```

```
Mergesort: visualization
```

```
Mergesort: visualization
```

```
Mergesort: visualization
```

```
Mergesort: visualization
```

```
Mergesort: visualization
```

```
Mergesort: visualization
```

```
Mergesort: visualization
```

```
Mergesort: visualization
```

```
Mergesort: visualization
```

```
Mergesort: visualization
```

```
Mergesort: visualization
```

```
Mergesort: visualization
```

```
Mergesort: visualization
```

```
Mergesort: visualization
```

```
Mergesort: visualization
```

```
Mergesort: visualization
```

```
Mergesort: visualization
```

```
Mergesort: visualization
```

```
Mergesort: visualization
```

```
Mergesort: visualization
```

```
Mergesort: visualization
```

```
Mergesort: visualization
```

```
Mergesort: visualization
```

```
Mergesort: visualization
```

```
Mergesort: visualization
```

```
Mergesort: visualization
```

```
Mergesort: visualization
```

```
Mergesort: visualization
```

```
Mergesort: visualization
```

```
Mergesort: visualization
```

```
Mergesort: visualization
```

```
Mergesort: visualization
```

```
Mergesort: visualization
```

```
Mergesort: visualization
```

```
Mergesort: visualization
```

```
Mergesort: visualization
```

```
Mergesort: visualization
```

```
Mergesort: visualization
```

```
Mergesort: visualization
```

```
Mergesort: visualization
```

```
Mergesort: visualization
```

```
Mergesort: visualization
```

```
Mergesort: visualization
```

```
Mergesort: visualization
```

```
Mergesort: visualization
```

```
Mergesort: visualization
```

```
Mergesort: visualization
```

```
Mergesort: visualization
```

```
Mergesort: visualization
```

```
Mergesort: visualization
```

```
Mergesort: visualization
```

```
Mergesort: visualization
```

```
Mergesort: visualization
```

```
Mergesort: visualization
```

```
Mergesort: visualization
```

```
Mergesort: visualization
```

```
Mergesort: visualization
```

```
Mergesort: visualization
```

```
Mergesort: visualization
```

```
Mergesort: visualization
```
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

http://bl.ocks.org/mbostock/39566aca95eb03ddd526

http://bl.ocks.org/mbostock/e65d9895da07c57e94bd
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: ?

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.

Decision tree (for 3 distinct items a, b, and c)

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.

$2^h \geq N! \Rightarrow h \geq \lg (N!) \sim N \lg N$
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: ~ N lg N from mergesort.
• Lower bound: ~ 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 ½ 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.

Sort music library by artist name
Sort music library by song name

Comparable interface: review

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

```java
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: system sort

To use with Java system sort:

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

```java
String[] a; uses natural order
... Arrays.sort(a); uses alternate order defined by Comparator<String> object
... 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

Comparator interface: sort using an alternate order.

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

Required property. Must be a total order.

Ex. Sort strings by:

• Natural order.
• Case insensitive.
• Spanish.
• British phone book.
• Pre-1994 order for digraphs ch and ll and rr.
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()`.

```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;  }
    }
}
```

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.

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

**Drawback.** Evaluating a trigonometric function is expensive.
Polar order

Given a point \( p \), order points by the polar angle \( \theta \) 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, \( \text{ccw}(p, q_1, q_2) \) identifies which of \( q_1 \) or \( q_2 \) makes larger polar angle.

Comparator interface: polar order

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

Stability

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

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

@#$%@$ Students in section 3 no longer sorted by name.
Stability: insertion sort

**Proposition.** Insertion sort is stable.

```java
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);
    }
}
```

Stability: selection sort

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

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

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

Stability: shellsort

**Proposition.** Shellsort sort is not stable.

```java
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;
    }
}
```

Stability: mergesort

**Proposition.** Mergesort is stable.

**Pf.** Suffices to verify that merge operation is stable.

```java
public class Merge {
    private static Comparable[] aux;
    private static void merge(Comparable[] a, int lo, int mid, int hi) {
        /* as before */
    }
    private static void sort(Comparable[] a, int lo, int hi) {
        if (hi <= lo) return;
        int mid = lo + (hi - lo) / 2;
        sort(a, lo, mid);
        sort(a, mid+1, hi);
        merge(a, lo, mid, hi);
    }
    public static void sort(Comparable[] a) {
        /* as before */
    }
}
```
Proposition. Merge operation is stable.

Pf. Takes from left subarray if equal keys.

```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++];
    }
}
```