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.
• Java sort for objects.
• Perl, C++ stable sort, Python stable sort, Firefox JavaScript, ...

Quicksort.
• 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
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>a[]</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>aux[]</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>i</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>j</td>
<td>0</td>
<td>6</td>
<td>0</td>
<td>7</td>
<td>1</td>
<td>7</td>
<td>2</td>
<td>7</td>
<td>2</td>
<td>8</td>
</tr>
</tbody>
</table>

| merged result | A  | C  | E  | E  | E  | G  | M  | R  | R  | T  |

Abstract in-place merge trace
Merging: Java implementation

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
}

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

<table>
<thead>
<tr>
<th>aux[]</th>
<th>A</th>
<th>G</th>
<th>L</th>
<th>O</th>
<th>R</th>
<th>H</th>
<th>I</th>
<th>M</th>
<th>S</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>a[]</td>
<td>A</td>
<td>G</td>
<td>H</td>
<td>I</td>
<td>L</td>
<td></td>
<td></td>
<td>M</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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).
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

Trace of merge results for top-down mergesort

result after recursive call
Mergesort animation

50 random elements

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

50 reverse-sorted elements

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 showing comparison of running times between insertion sort ($N^2$) and mergesort ($N \log N$) for different computers.]

**Bottom line.** Good algorithms are better than supercomputers.
**Proposition.** Mergesort uses at most $N \log N$ compares and $6N \log N$ 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 $N$ satisfies the recurrences:

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

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

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

$$D(N) = 2D\left(\frac{N}{2}\right) + N, \text{ for } N > 1, \text{ with } D(1) = 0.$$  

*result holds for all $N$ (see COS 340)*
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]

\[
\begin{align*}
D(N) & = 2D(N/2) + N \\
D(N) / N & = 2D(N/2) / N + 1 \\
& = D(N/2) / (N/2) + 1 \\
& = D(N/4) / (N/4) + 1 + 1 \\
& = D(N/8) / (N/8) + 1 + 1 + 1 \\
& \vdots \\
& = D(N/N) / (N/N) + 1 + 1 + \ldots + 1 \\
& = \lg N
\end{align*}
\]
Proposition. If \( D(N) \) satisfies \( D(N) = 2 \, D(N/2) + N \) 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 \( D(2N) = (2N) \lg (2N) \).

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

given

inductive hypothesis

algebra

QED
**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 $O(\log N)$ 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 $\approx 7$ elements.

```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 element in first half ≤ smallest element 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;
   if (!less(a[mid+1], a[mid])) return;
   sort (a, aux, lo, mid);
   sort (a, aux, mid+1, hi);
   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);
}
```
Mergesort: practical improvements

Use insertion sort for small subarrays.
• Mergesort has too much overhead for tiny subarrays.
• Cutoff to insertion sort for $\approx 7$ elements.

Stop if already sorted.
• Is biggest element in first half $\leq$ smallest element in second half?
• Helps for partially-ordered arrays.

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.

Ex. See MergeX.java or Arrays.sort().
Mergesort visualization

- 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, ....

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.** Specify allowable operations.

**Cost model.** Focus on fundamental operations.

**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 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 ???

lower bound $\sim$ upper bound

can access information only through compares (e.g., our Java sorting framework)
Decision tree (for 3 distinct elements 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
        - b c a

- **b RxRc**
  - yes
    - **a RxRc**
      - yes
        - a b c
      - no
        - 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.
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. Specify allowable operations.
Cost model. Focus on fundamental operations.
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 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 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 $\frac{1}{2}N \log N$ 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 by artist name

<table>
<thead>
<tr>
<th>Name</th>
<th>Artist</th>
<th>Time</th>
<th>Album</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>Let It Be</td>
<td>The Beatles</td>
<td>4:03</td>
</tr>
<tr>
<td>13</td>
<td>Take My Breath Away</td>
<td>BERLIN</td>
<td>4:13</td>
</tr>
<tr>
<td>14</td>
<td>Circle Of Friends</td>
<td>Better Than Ezra</td>
<td>3:27</td>
</tr>
<tr>
<td>15</td>
<td>Dancing With Myself</td>
<td>Billy Idol</td>
<td>4:43</td>
</tr>
<tr>
<td>16</td>
<td>Rebel Yell</td>
<td>Billy Idol</td>
<td>4:49</td>
</tr>
<tr>
<td>17</td>
<td>Piano Man</td>
<td>Billy Joel</td>
<td>3:36</td>
</tr>
<tr>
<td>20</td>
<td>Atomic</td>
<td>Blondie</td>
<td>3:50</td>
</tr>
<tr>
<td>21</td>
<td>Sunday Girl</td>
<td>Blondie</td>
<td>3:15</td>
</tr>
<tr>
<td>22</td>
<td>Call Me</td>
<td>Blondie</td>
<td>3:33</td>
</tr>
<tr>
<td>23</td>
<td>Dreaming</td>
<td>Blondie</td>
<td>3:06</td>
</tr>
<tr>
<td>24</td>
<td>Hurricane</td>
<td>Bob Dylan</td>
<td>8:32</td>
</tr>
<tr>
<td>25</td>
<td>The Times They Are A-Changin’</td>
<td>Bob Dylan</td>
<td>3:17</td>
</tr>
<tr>
<td>26</td>
<td>Livin’ On A Prayer</td>
<td>Bon Jovi</td>
<td>4:11</td>
</tr>
<tr>
<td>27</td>
<td>Beds Of Roses</td>
<td>Bon Jovi</td>
<td>6:35</td>
</tr>
<tr>
<td>28</td>
<td>Runaway</td>
<td>Bon Jovi</td>
<td>3:53</td>
</tr>
<tr>
<td>29</td>
<td>Rasputin (Extended Mix)</td>
<td>Boney M</td>
<td>5:50</td>
</tr>
<tr>
<td>30</td>
<td>Have You Ever Seen The Rain</td>
<td>Bonnie Tyler</td>
<td>4:10</td>
</tr>
<tr>
<td>31</td>
<td>Total Eclipse Of The Heart</td>
<td>Bonnie Tyler</td>
<td>7:02</td>
</tr>
<tr>
<td>32</td>
<td>Straight From The Heart</td>
<td>Bonnie Tyler</td>
<td>3:41</td>
</tr>
<tr>
<td>33</td>
<td>Holding Out For A Hero</td>
<td>Bonny Tyler</td>
<td>5:49</td>
</tr>
<tr>
<td>34</td>
<td>Dancing In The Dark</td>
<td>Bruce Springsteen</td>
<td>4:51</td>
</tr>
<tr>
<td>35</td>
<td>Thunder Road</td>
<td>Bruce Springsteen</td>
<td>4:30</td>
</tr>
<tr>
<td>36</td>
<td>Born To Run</td>
<td>Bruce Springsteen</td>
<td>9:34</td>
</tr>
<tr>
<td>37</td>
<td>Jungleland</td>
<td>Bruce Springsteen</td>
<td>3:53</td>
</tr>
<tr>
<td>38</td>
<td>Born To Run</td>
<td>Bruce Springsteen</td>
<td>9:34</td>
</tr>
</tbody>
</table>
Sort 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>Amia, Amia, Amia Y Ensancha El...</td>
<td></td>
<td>2:34</td>
<td>Deluca (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 Johanson</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>Insomniac</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></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 – The Collection</td>
</tr>
<tr>
<td>Chasing Changes</td>
<td>Subraders Spot</td>
<td>5:11</td>
<td>Bon Jovi: Dance</td>
</tr>
</tbody>
</table>
Comparable interface: sort uses 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;
    }
}
```
Generalized compare

**Comparable interface:** sort uses type's natural order.

**Problem 1.** May want to use a non-natural order.
**Problem 2.** Desired data type may not come with a “natural” 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

```java
String[] a;
...
Arrays.sort(a);
Arrays.sort(a, String.CASE_INSENSITIVE_ORDER);
Arrays.sort(a, Collator.getInstance(Locale.SPANISH));
import java.text.Collator;
```
Comparators

Solution. Use Java's `Comparator` interface.

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

Remark. `compare()` must implement a total order like `compareTo()`.

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

- Can add any number of new orders to a data type.
- Can add an order to a library data type with no natural order.
Comparator example

Reverse order. Sort an array of strings in reverse order.

```java
public class ReverseOrder implements Comparator<String> {
    public int compare(String a, String b) {
        return b.compareTo(a);
    }
}
```

```java
Arrays.sort(a, new ReverseOrder());
```

comparator implementation

client
Sort implementation with comparators

To support comparators in our sort implementations:
- Use Object instead of Comparable.
- Pass Comparator to sort() and less().
- Use it in less().

Ex. Insertion sort.

```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; }
```
Generalized compare

Comparators enable multiple sorts of a single array (by different keys).

**Ex.** Sort students by name or by section.

```java
Arrays.sort(students, Student.BY_NAME);
Arrays.sort(students, Student.BY_SECT);
```

<table>
<thead>
<tr>
<th>Name</th>
<th>Group</th>
<th>Phone Number</th>
<th>Room</th>
</tr>
</thead>
<tbody>
<tr>
<td>Andrews</td>
<td>3</td>
<td>664-480-0023</td>
<td>097 Little</td>
</tr>
<tr>
<td>Battle</td>
<td>4</td>
<td>874-088-1212</td>
<td>121 Whitman</td>
</tr>
<tr>
<td>Chen</td>
<td>2</td>
<td>991-878-4944</td>
<td>308 Blair</td>
</tr>
<tr>
<td>Fox</td>
<td>1</td>
<td>884-232-5341</td>
<td>11 Dickinson</td>
</tr>
<tr>
<td>Furia</td>
<td>3</td>
<td>766-093-9873</td>
<td>101 Brown</td>
</tr>
<tr>
<td>Gazsi</td>
<td>4</td>
<td>665-303-0266</td>
<td>22 Brown</td>
</tr>
<tr>
<td>Kanaga</td>
<td>3</td>
<td>898-122-9643</td>
<td>22 Brown</td>
</tr>
<tr>
<td>Rohde</td>
<td>3</td>
<td>232-343-5555</td>
<td>343 Forbes</td>
</tr>
</tbody>
</table>

**sort by name**

<table>
<thead>
<tr>
<th>Name</th>
<th>Group</th>
<th>Phone Number</th>
<th>Room</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fox</td>
<td>1</td>
<td>884-232-5341</td>
<td>11 Dickinson</td>
</tr>
<tr>
<td>Chen</td>
<td>2</td>
<td>991-878-4944</td>
<td>308 Blair</td>
</tr>
<tr>
<td>Andrews</td>
<td>3</td>
<td>664-480-0023</td>
<td>097 Little</td>
</tr>
<tr>
<td>Furia</td>
<td>3</td>
<td>766-093-9873</td>
<td>101 Brown</td>
</tr>
<tr>
<td>Kanaga</td>
<td>3</td>
<td>898-122-9643</td>
<td>22 Brown</td>
</tr>
<tr>
<td>Rohde</td>
<td>3</td>
<td>232-343-5555</td>
<td>343 Forbes</td>
</tr>
<tr>
<td>Battle</td>
<td>4</td>
<td>874-088-1212</td>
<td>121 Whitman</td>
</tr>
<tr>
<td>Gazsi</td>
<td>4</td>
<td>665-303-0266</td>
<td>22 Brown</td>
</tr>
</tbody>
</table>
Generalized compare

Ex. Enable sorting students by name or by section.

```java
public class Student {

    public static final Comparator<Student> BY_NAME = new ByName();
    public static final Comparator<Student> BY_SECT = new BySect();

    private final String name;
    private final int section;
    ...

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

    private static class BySect implements Comparator<Student> {
        public int compare(Student a, Student b) {
            return a.section - b.section;
        }
    }

    // use this trick only if no danger of overflow
```
- mergesort
- bottom-up mergesort
- sorting complexity
- comparators
- stability
**Generalized compare problem**

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

Arrays.sort(students, Student.BY_NAME);

<table>
<thead>
<tr>
<th>Name</th>
<th>Sect</th>
<th>Number</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>2</td>
<td>A</td>
<td>991-878-4944</td>
<td>308 Blair</td>
</tr>
<tr>
<td>Fox</td>
<td>1</td>
<td>A</td>
<td>884-232-5341</td>
<td>11 Dickinson</td>
</tr>
<tr>
<td>Furia</td>
<td>3</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>665-303-0266</td>
<td>22 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>3</td>
<td>A</td>
<td>232-343-5555</td>
<td>343 Forbes</td>
</tr>
</tbody>
</table>

Arrays.sort(students, Student.BY_SECT);

<table>
<thead>
<tr>
<th>Name</th>
<th>Sect</th>
<th>Number</th>
<th>Phone</th>
<th>Room</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fox</td>
<td>1</td>
<td>A</td>
<td>884-232-5341</td>
<td>11 Dickinson</td>
</tr>
<tr>
<td>Chen</td>
<td>2</td>
<td>A</td>
<td>991-878-4944</td>
<td>308 Blair</td>
</tr>
<tr>
<td>Kanaga</td>
<td>3</td>
<td>B</td>
<td>898-122-9643</td>
<td>22 Brown</td>
</tr>
<tr>
<td>Andrews</td>
<td>3</td>
<td>A</td>
<td>664-480-0023</td>
<td>097 Little</td>
</tr>
<tr>
<td>Furia</td>
<td>3</td>
<td>A</td>
<td>766-093-9873</td>
<td>101 Brown</td>
</tr>
<tr>
<td>Rohde</td>
<td>3</td>
<td>A</td>
<td>232-343-5555</td>
<td>343 Forbes</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>665-303-0266</td>
<td>22 Brown</td>
</tr>
</tbody>
</table>

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

**A stable sort preserves the relative order of records with equal keys.**
### Q. Which sorts are stable?

**Insertion sort?**  **Selection sort?**  **Shellsort?**  **Mergesort?**

<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 09:21:05</td>
<td>Chicago 09:00:59</td>
</tr>
<tr>
<td>Houston 09:00:13</td>
<td>Chicago 09:19:46</td>
<td>Chicago 09:19:32</td>
</tr>
<tr>
<td>Chicago 09:00:59</td>
<td>Chicago 09:19:32</td>
<td>Chicago 09:19:46</td>
</tr>
<tr>
<td>Houston 09:01:10</td>
<td>Chicago 09:00:00</td>
<td>Chicago 09:21:05</td>
</tr>
<tr>
<td>Chicago 09:03:13</td>
<td>Chicago 09:35:21</td>
<td>Chicago 09:25:52</td>
</tr>
<tr>
<td>Seattle 09:10:11</td>
<td>Chicago 09:00:59</td>
<td>Chicago 09:35:21</td>
</tr>
<tr>
<td>Seattle 09:10:25</td>
<td>Chicago 09:00:13</td>
<td>Houston 09:00:13</td>
</tr>
<tr>
<td>Phoenix 09:14:25</td>
<td>Houston 09:00:13</td>
<td>Phoenix 09:00:03</td>
</tr>
<tr>
<td>Chicago 09:19:46</td>
<td>Phoenix 09:37:44</td>
<td>Phoenix 09:37:44</td>
</tr>
<tr>
<td>Chicago 09:21:05</td>
<td>Seattle 09:10:25</td>
<td>Seattle 09:10:11</td>
</tr>
<tr>
<td>Seattle 09:22:43</td>
<td>Seattle 09:36:14</td>
<td>Seattle 09:10:25</td>
</tr>
<tr>
<td>Seattle 09:22:54</td>
<td>Seattle 09:36:14</td>
<td>Seattle 09:22:43</td>
</tr>
<tr>
<td>Seattle 09:35:21</td>
<td>Seattle 09:10:11</td>
<td>Seattle 09:22:54</td>
</tr>
<tr>
<td>Seattle 09:36:14</td>
<td>Seattle 09:22:54</td>
<td>Seattle 09:36:14</td>
</tr>
</tbody>
</table>

**Stability when sorting on a second key**
Q. Is insertion sort stable?

A. Yes, equal elements never more past each other.
**Q. Is selection sort stable?**

**A. No, long-distance exchange might move left element to the right of some equal element.**
Sorting challenge 5C

Q. Is shellsort stable?

A. No. Long-distance exchanges.

```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;
            }
        }
    }
}
```
Q. Is mergesort 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 */ }
}
```
Q. Is mergesort stable?

**A.** Yes, if merge is stable.

Trace of merge results for bottom-up mergesort
Q. Is merge stable?

A. Yes, if implemented carefully (take from left subarray if equal).

```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++];
    }
}
```
Sorting challenge 5 (summary)

Q. Which sorts are stable?

Yes. Insertion sort, mergesort.
No. Selection sort, shellsort.

Note. Need to carefully check code ("less than" vs "less than or equal to").
Postscript: optimizing mergesort (a short history)

**Goal.** Remove instructions from the inner loop.

```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++];
}
```
Postscript: optimizing mergesort (a short history)


```c
a[M] := maxint; b[N] := maxint;
for (int i = 0, j = 0, k = 0; k < M+1; k++)
   if (less(aux[j], aux[i])) aux[k] = a[i++];
   aux[k] = b[j++];
```

Problem 1. Still need copy.
Problem 2. No good place to put sentinels.
Problem 3. Complicates data-type interface (what is infinity for your type?)
Postscript: optimizing mergesort (a short history)


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

    for (int j = mid+1; j <= hi; j++)
        aux[j] = a[hi-j+mid+1]; // reverse copy

    int i = lo, j = hi;
    for (int k = lo; k <= hi; k++)
        if (less(aux[j], aux[i])) a[k] = aux[j--];
        else a[k] = aux[i++]; // merge

}
```

Problem. Copy still in inner loop.
Postscript: optimizing mergesort (a short history)


```java
int mid = (lo+hi)/2;
mergesortABr(b, a, lo, mid);
mergesortABr(b, a, mid+1, hi);
mergeAB(a, lo, b, lo, mid, b, mid+1, hi);
```

Problem. Complex interactions with reverse copy.
Solution. Go back to sentinels.

```java
Arrays.sort()
```
Problem. Choose mergesort for Algs 4th edition. Recursive argument switch is out (recommended only for pros).

Q. Why not use reverse array copy?

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

    for (int j = mid+1; j <= hi; j++)
        aux[j] = a[hi-j+mid+1];

    int i = lo, j = hi;
    for (int k = lo; k <= hi; k++)
        if (less(aux[j], aux[i])) a[k] = aux[j--];
        else                      a[k] = aux[i++];
}
```
Sorting challenge 6

Recursive argument switch is out (recommended only for pros)

Q. Why not use reverse array copy?

A. It is not stable (!)

Solution. Back to the standard algorithm!