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, ...
2.2 MERGESORT

- 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**

<table>
<thead>
<tr>
<th>input</th>
<th>M E R G E S O R T E X A M P L E</th>
</tr>
</thead>
<tbody>
<tr>
<td>sort left half</td>
<td>E E G M O R R S T E X A M P L E</td>
</tr>
<tr>
<td>sort right half</td>
<td>E E G M O R R S A E E L M P T X</td>
</tr>
<tr>
<td>merge results</td>
<td>A E E E E E G L M M O P R R S T X</td>
</tr>
</tbody>
</table>
**Goal.** Given two sorted subarrays $a[lo]$ to $a[mid]$ and $a[mid+1]$ to $a[hi]$, replace with sorted subarray $a[lo]$ to $a[hi]$.

<table>
<thead>
<tr>
<th></th>
<th>lo</th>
<th>mid</th>
<th>mid+1</th>
<th>hi</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a[]$</td>
<td>E</td>
<td>E</td>
<td>G</td>
<td>M</td>
</tr>
</tbody>
</table>

*sorted*
Abstract in-place merge demo

**Goal.** Given two sorted subarrays \(a[lo]\) to \(a[mid]\) and \(a[mid+1]\) to \(a[hi]\), replace with sorted subarray \(a[lo]\) to \(a[hi]\).
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 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 assertions to check internal invariants;
assume assertions will be disabled in production code. → do not use for external argument checking
Mergesort: Java implementation

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

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

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

merge(a, aux, 0, 0, 1)
merge(a, aux, 2, 2, 3)
merge(a, aux, 0, 1, 3)
merge(a, aux, 4, 4, 5)
merge(a, aux, 6, 6, 7)
merge(a, aux, 4, 5, 7)
merge(a, aux, 0, 3, 7)
merge(a, aux, 8, 8, 9)
merge(a, aux, 10, 10, 11)
merge(a, aux, 8, 9, 11)
merge(a, aux, 12, 12, 13)
merge(a, aux, 14, 14, 15)
merge(a, aux, 12, 13, 15)
merge(a, aux, 8, 11, 15)
merge(a, aux, 0, 7, 15)

\[
\begin{array}{cccccccccccccccc}
\text{lo} & & & & & & & & & & & & & & & & \\
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 \\
\text{hi} & & & & & & & & & & & & & & & & \\
\end{array}
\]

result after recursive call

a[]

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.

<table>
<thead>
<tr>
<th>computer</th>
<th>insertion sort (N²)</th>
<th>mergesort (N log N)</th>
</tr>
</thead>
<tbody>
<tr>
<td>home</td>
<td>instant</td>
<td>instant</td>
</tr>
<tr>
<td></td>
<td>2.8 hours</td>
<td>1 second</td>
</tr>
<tr>
<td></td>
<td>317 years</td>
<td>18 min</td>
</tr>
<tr>
<td>super</td>
<td>instant</td>
<td>instant</td>
</tr>
<tr>
<td></td>
<td>1 second</td>
<td>instant</td>
</tr>
<tr>
<td></td>
<td>1 week</td>
<td>instant</td>
</tr>
</tbody>
</table>

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:

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

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

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]

\[
\begin{align*}
D(N) &= N \\
2(N/2) &= N \\
4(N/4) &= N \\
& \vdots \\
2^k(N/2^k) &= N \\
N/2(2) &= N \\
\hline
N \lg N
\end{align*}
\]
**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]

\[
\begin{align*}
D(N) &= 2D(N/2) + N \\
D(N)/N &= 2D(N/2)/N + 1 \\
    &= D(N/2)/(N/2) + 1 \\
    &= D(N/4)/(N/4) + 1 + 1 \\
    &= D(N/8)/(N/8) + 1 + 1 + 1 \\
    &\ldots \\
    &= D(N/N)/(N/N) + 1 + 1 + \ldots + 1 \\
    &= \lg N
\end{align*}
\]
Divide-and-conquer recurrence: proof by induction

**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
**Mergesort analysis: memory**

**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 \( \leq c \log N \) 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);
        return;
    }
    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.

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(a, aux, lo, mid, hi);
}
```

Note: sort(a) initializes aux[] and sets aux[i] = a[i] for each i.
Mergesort: visualization

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

- mergesort
- bottom-up mergesort
- sorting complexity
- comparators
- stability
2.2 Mergesort

- 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>
<th>a[i]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
</tr>
<tr>
<td>merge(a, aux, 0, 0, 1)</td>
<td>E M R G E S O R T E X A M P L E</td>
</tr>
<tr>
<td>merge(a, aux, 2, 2, 3)</td>
<td>E M G R E S O R T E X A M P L E</td>
</tr>
<tr>
<td>merge(a, aux, 4, 4, 5)</td>
<td>E M G R E S O R T E X A M P L E</td>
</tr>
<tr>
<td>merge(a, aux, 6, 6, 7)</td>
<td>E M G R E S O R T E X A M P L E</td>
</tr>
<tr>
<td>merge(a, aux, 8, 8, 9)</td>
<td>E M G R E S O R T E X A M P L E</td>
</tr>
<tr>
<td>merge(a, aux, 10, 10, 11)</td>
<td>E M G R E S O R T E X A M P L E</td>
</tr>
<tr>
<td>merge(a, aux, 12, 12, 13)</td>
<td>E M G R E S O R T E X A M P L E</td>
</tr>
<tr>
<td>merge(a, aux, 14, 14, 15)</td>
<td>E M G R E S O R T E X A M P L E</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>sz = 2</th>
<th>a[i]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
</tr>
<tr>
<td>merge(a, aux, 0, 1, 3)</td>
<td>E G M R E S O R T E X A M P L E</td>
</tr>
<tr>
<td>merge(a, aux, 4, 5, 7)</td>
<td>E G M R E O R S E T A X M P L E</td>
</tr>
<tr>
<td>merge(a, aux, 8, 9, 11)</td>
<td>E G M R E O R S A E T X M P L E</td>
</tr>
<tr>
<td>merge(a, aux, 12, 13, 15)</td>
<td>E G M R E O R S A E T X E L M P</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>sz = 4</th>
<th>a[i]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
</tr>
<tr>
<td>merge(a, aux, 0, 3, 7)</td>
<td>E E G M O R R S A E T X E L M P</td>
</tr>
<tr>
<td>merge(a, aux, 8, 11, 15)</td>
<td>E E G M O R R S A E E L M P T X</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>sz = 8</th>
<th>a[i]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
</tr>
<tr>
<td>merge(a, aux, 0, 7, 15)</td>
<td>A E E E E E G L M M O P R R S T X</td>
</tr>
</tbody>
</table>
Bottom-up mergesort: Java implementation

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

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

but about 10% slower than recursive, top-down mergesort on typical systems

Bottom line. Simple and non-recursive version of mergesort.
Bottom-up mergesort: visual trace
2.2 Mergesort

- mergesort
- bottom-up mergesort
- sorting complexity
- comparators
- stability
2.2 MERGESORT

- 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: $\ ?$

Lower bound $\sim$ upper bound

Can access information only through compares (e.g., Java Comparable framework)
Decision tree (for 3 distinct items a, b, and c)

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

a < b
  yes
  b < c
    yes
      a b c
    no
      a < c
        yes
          a c b
        no
          c a b
  no
    a < c
      yes
        a c b
      no
        c a b

b < c
  yes
  a < c
    yes
      a b c
    no
      b a c
  no
    b < c
      yes
        b c a
      no
        c b a

height of tree = worst-case number of compares

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

(at least) one leaf for each possible ordering)
```
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.

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

**Compares?** Mergesort is optimal with respect to number compares.
**Space?** Mergesort is not optimal with respect to space usage.

**Lessons.** Use theory as a guide.
**Ex.** Design sorting algorithm that guarantees $\frac{1}{2} N \lg N$ compares?
**Ex.** Design sorting algorithm that is both time- and space-optimal?
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.
2.2 Mergesort

- mergesort
- bottom-up mergesort
- sorting complexity
- comparators
- stability
2.2 Mergesort

- mergesort
- bottom-up mergesort
- sorting complexity
- comparators
- stability
Sort music library by artist name
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>Ama, Ama, Ama Y Ensancha El...</td>
<td>Extremoduro</td>
<td>2:34</td>
<td>Deltoya (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>Brut</td>
<td>Green Day</td>
<td>1:43</td>
<td>Insomniac</td>
</tr>
<tr>
<td>Breakdown</td>
<td>Deedorth</td>
<td>3:40</td>
<td>Doorheart</td>
</tr>
<tr>
<td>Bring Me To Life (Kevin Roen Mix)</td>
<td>Evanescence Vs. Pa...</td>
<td>6:48</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>Kylic 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 – 1970’s</td>
</tr>
<tr>
<td>Chains Of Chains</td>
<td>Eddy Under 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.

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

```java
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);
}

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;
}
```
To implement a comparator:

- Define a (nested) class that implements the Comparator interface.
- Implement the `compareTo()` 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);
        }
    }

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

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></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 polar angle they make with \( p \).

\[
\begin{array}{cccccccc}
3 & 2 & 0 & 1 & 4 & 5 & 6 & 7 \\
\end{array}
\]

\[
\text{Arrays.sort(points, p.POLAR_ORDER);} \;
\]

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 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 angle.

```java
Arrays.sort(points, p.POLAR_ORDER);
```
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 dy1 = q1.y - y;
            double dy2 = q2.y - y;

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

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

- mergesort
- bottom-up mergesort
- sorting complexity
- comparators
- stability
2.2 Mergesort

- 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);

<table>
<thead>
<tr>
<th>Name</th>
<th>Section</th>
<th>Phone</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>Andrews</td>
<td>3</td>
<td>A</td>
<td>664-480-0023</td>
</tr>
<tr>
<td>Battle</td>
<td>4</td>
<td>C</td>
<td>874-088-1212</td>
</tr>
<tr>
<td>Chen</td>
<td>3</td>
<td>A</td>
<td>991-878-4944</td>
</tr>
<tr>
<td>Fox</td>
<td>3</td>
<td>A</td>
<td>884-232-5341</td>
</tr>
<tr>
<td>Furia</td>
<td>1</td>
<td>A</td>
<td>766-093-9873</td>
</tr>
<tr>
<td>Gazsi</td>
<td>4</td>
<td>B</td>
<td>766-093-9873</td>
</tr>
<tr>
<td>Kanaga</td>
<td>3</td>
<td>B</td>
<td>898-122-9643</td>
</tr>
<tr>
<td>Rohde</td>
<td>2</td>
<td>A</td>
<td>232-343-5555</td>
</tr>
</tbody>
</table>

Selection.sort(a, Student.BY_SECTION);

<table>
<thead>
<tr>
<th>Name</th>
<th>Section</th>
<th>Phone</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>Furia</td>
<td>1</td>
<td>A</td>
<td>766-093-9873</td>
</tr>
<tr>
<td>Rohde</td>
<td>2</td>
<td>A</td>
<td>232-343-5555</td>
</tr>
<tr>
<td>Chen</td>
<td>3</td>
<td>A</td>
<td>991-878-4944</td>
</tr>
<tr>
<td>Fox</td>
<td>3</td>
<td>A</td>
<td>884-232-5341</td>
</tr>
<tr>
<td>Andrews</td>
<td>3</td>
<td>A</td>
<td>664-480-0023</td>
</tr>
<tr>
<td>Kanaga</td>
<td>3</td>
<td>B</td>
<td>898-122-9643</td>
</tr>
<tr>
<td>Gazsi</td>
<td>4</td>
<td>B</td>
<td>766-093-9873</td>
</tr>
<tr>
<td>Battle</td>
<td>4</td>
<td>C</td>
<td>874-088-1212</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 09:19:00</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>Houston 09:01:10</td>
<td>Chicago 09:00:13</td>
</tr>
<tr>
<td>Phoenix 09:14:25</td>
<td>Houston 09:00:13</td>
<td>Houston 09:01:10</td>
</tr>
<tr>
<td>Chicago 09:19:32</td>
<td>Phoenix 09:37:44</td>
<td>Phoenix 09:00:03</td>
</tr>
<tr>
<td>Chicago 09:19:46</td>
<td>Phoenix 09:00:03</td>
<td>Phoenix 09:14:25</td>
</tr>
<tr>
<td>Chicago 09:21:05</td>
<td>Phoenix 09:14:25</td>
<td>Phoenix 09:37:44</td>
</tr>
<tr>
<td>Seattle 09:22:43</td>
<td>Phoenix 09:14:25</td>
<td>Seattle 09:10:11</td>
</tr>
<tr>
<td>Seattle 09:22:54</td>
<td>Seattle 09:10:25</td>
<td>Seattle 09:10:25</td>
</tr>
<tr>
<td>Chicago 09:25:52</td>
<td>Seattle 09:36:14</td>
<td>Seattle 09:22:43</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").
**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);
    }
}
```

<table>
<thead>
<tr>
<th>i</th>
<th>j</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>B₁</td>
<td>A₁</td>
<td>A₂</td>
<td>A₃</td>
<td>B₂</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>A₁</td>
<td>B₁</td>
<td>A₂</td>
<td>A₃</td>
<td>B₂</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>A₁</td>
<td>A₂</td>
<td>B₁</td>
<td>A₃</td>
<td>B₂</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>A₁</td>
<td>A₂</td>
<td>A₃</td>
<td>B₁</td>
<td>B₂</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>A₁</td>
<td>A₂</td>
<td>A₃</td>
<td>B₁</td>
<td>B₂</td>
</tr>
</tbody>
</table>

**Pf.** Equal items never move past each other.
Proposition. Selection sort is not stable.

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

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

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

**Proposition.** Mergesort 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 */ }
}
```

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

```
private static void merge(...)
{
    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.
2.2 Mergesort

- mergesort
- bottom-up mergesort
- sorting complexity
- comparators
- stability
2.2 MERGESORT

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