Mergesort

Feb. 23, 2017

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

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 $a[hi]$.

<table>
<thead>
<tr>
<th>lo</th>
<th>mid</th>
<th>mid+1</th>
<th>hi</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>E</td>
<td>G</td>
<td>M</td>
</tr>
<tr>
<td>A</td>
<td>C</td>
<td>E</td>
<td>R</td>
</tr>
</tbody>
</table>

Mergesort overview

First Draft of a Report on the EDVAC

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 $a[hi]$.

<table>
<thead>
<tr>
<th>lo</th>
<th>mid</th>
<th>mid+1</th>
<th>hi</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>E</td>
<td>G</td>
<td>M</td>
</tr>
<tr>
<td>A</td>
<td>C</td>
<td>E</td>
<td>R</td>
</tr>
</tbody>
</table>

copy to auxiliary array

aux[]
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

compare minimum in each subarray

$\text{aux}[i..j]$
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]`.

```plaintext
<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>C</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>k</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

compare minimum in each subarray

```plaintext
<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>i</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>j</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]`.

```plaintext
<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>C</th>
<th>E</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>k</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

compare minimum in each subarray

```plaintext
<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>i</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>j</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 $a[hi]$.

$$a[]: \quad A \quad C \quad E \quad E \quad R \quad A \quad C \quad E \quad R \quad T \quad k$$

compare minimum in each subarray

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

**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 $a[hi]$.

$$a[]: \quad A \quad C \quad E \quad E \quad E \quad A \quad C \quad E \quad R \quad T \quad k$$

compare minimum in each subarray

$$aux[]: \quad E \quad E \quad G \quad M \quad R \quad A \quad C \quad E \quad R \quad T \quad i \quad j$$
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 $a[hi]$.

```
   a[]  A  C  E  E  E  G  C  E  R  T
              k
```

compare minimum in each subarray

```
   aux[] E  E  G  M  R  A  C  E  R  T
        i    j
```

17

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[]  A  C  E  E  E  G  M  E  R  T
              k
```

compare minimum in each subarray

```
   aux[] E  E  G  M  R  A  C  E  R  T
        i    j
```

18

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[]  A  C  E  E  E  G  M  E  R  T
              k
```

compare minimum in each subarray

```
   aux[] E  E  G  M  R  A  C  E  R  T
        i    j
```

19

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[]  A  C  E  E  E  G  M  E  R  T
              k
```

compare minimum in each subarray

```
   aux[] E  E  G  M  R  A  C  E  R  T
        i    j
```

20
### 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 $a[hi]$.

![Diagram](image1)

- **compare minimum in each subarray**

![Diagram](image2)

- **one subarray exhausted, take from other**

![Diagram](image3)

- **one subarray exhausted, take from other**

![Diagram](image4)
### Abstract in-place merge

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

<table>
<thead>
<tr>
<th>( a[] )</th>
<th>A</th>
<th>C</th>
<th>E</th>
<th>E</th>
<th>E</th>
<th>G</th>
<th>M</th>
<th>R</th>
<th>R</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>( aux[] )</td>
<td>E</td>
<td>E</td>
<td>G</td>
<td>M</td>
<td>R</td>
<td>A</td>
<td>C</td>
<td>E</td>
<td>R</td>
<td>T</td>
</tr>
</tbody>
</table>

**one subarray exhausted, take from other**

<table>
<thead>
<tr>
<th>( aux[] )</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>( 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>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>
</tbody>
</table>

**both subarrays exhausted, done**

<table>
<thead>
<tr>
<th>( aux[] )</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>( i )</td>
<td>0</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( j )</td>
<td>0</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

### Merging

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

**A.** Use an auxiliary array.

| \( a[] \) | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 1 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| \( aux[] \) | E | E | G | M | R | A | C | E | R | T | E | E | G | M | R | A | C | E | R | T |
| \( k \) | 0 | 5 |
| \( i \) | 0 | 6 |
| \( j \) | 0 | 7 |

**merged result**

<table>
<thead>
<tr>
<th>A</th>
<th>C</th>
<th>E</th>
<th>E</th>
<th>E</th>
<th>G</th>
<th>M</th>
<th>R</th>
<th>R</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>( k )</td>
<td>0</td>
<td>5</td>
<td>6</td>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Abstract in-place merge trace
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
}
```

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>G</td>
<td>L</td>
<td>O</td>
<td>K</td>
</tr>
<tr>
<td>H</td>
<td>I</td>
<td>M</td>
<td>S</td>
<td>T</td>
</tr>
<tr>
<td>a[]</td>
<td>A</td>
<td>G</td>
<td>H</td>
<td>I</td>
</tr>
</tbody>
</table>
```

```
result after recursive call
```

Mergesort: animation

```
http://www.sorting-algorithms.com/merge-sort
```

50 random items
Mergesort: animation

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

Mergesort: number of compares and array accesses

**Proposition.** Mergesort uses at most \( N \lg N \) compares and \( 6 N \lg N \) 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:

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

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

We solve the recurrence when \( N \) is a power of 2.

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

Mergesort: empirical analysis

**Running time estimates:**
- Laptop executes \( 10^8 \) compares/second.
- Supercomputer executes \( 10^{12} \) compares/second.

<table>
<thead>
<tr>
<th>comparison type</th>
<th>computer</th>
<th>thousand</th>
<th>million</th>
<th>billion</th>
</tr>
</thead>
<tbody>
<tr>
<td>home</td>
<td>instant</td>
<td>2.8 hours</td>
<td>317 years</td>
<td>instant</td>
</tr>
<tr>
<td>super</td>
<td>instant</td>
<td>1 second</td>
<td>1 week</td>
<td>instant</td>
</tr>
</tbody>
</table>

**Bottom line.** Good algorithms are better than supercomputers.

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]

\[
D(N) = 2(N/2) = N
\]

\[
D(N/4) = 2(N/4) = N
\]

\[
D(N/8) = 2(N/8) = N
\]

\[
D(N/2^n) = 2(N/2^n) = N
\]

\[
N/2 = N
\]

\[
N \lg N
\]
**Divide-and-conquer recurrence: proof by expansion**

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

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

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

\( \text{given} \), divide both sides by \( N \), algebra, apply to first term, stop applying, \( D(1) = 0 \).

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

• **Base case:** \( N = 1 \).
• **Inductive hypothesis:** \( D(N) = N \lg N \).
• **Goal:** show that \( D(2N) = (2N) \lg (2N) \).

\[
\begin{align*}
D(2N) &= 2D(N) + 2N \\
&= 2N \lg N + 2N \\
&= 2N (\lg (2N) - 1) + 2N \\
&= 2N \lg (2N) \\
\end{align*}
\]

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 \( \leq c \log N \) 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.

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

Stop if already sorted.
• Is biggest item in first half ≤ smallest item in second half?
• Helps for partially-ordered arrays.

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

Mergesort: visualization

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

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

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

http://bl.ocks.org/mbostock/e65d9895da07c57e94bd
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. Lower bound \( \sim \) upper bound

- 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.

\[ \begin{align*}
2^h & \geq N! \\
\Rightarrow h & \geq \lg (N!) \sim N \lg N
\end{align*} \]

Stirling’s formula
**Complexity of sorting**

- **Model of computation.** Allowable operations.
- **Cost model.** Operation count(s).
- **Upper bound.** Cost guarantee provided by some algorithm for $X$.
- **Lower bound.** Proven limit on cost guarantee of all algorithms for $X$.
- **Optimal algorithm.** Algorithm with best possible cost guarantee for $X$.

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

First goal of algorithm design: optimal algorithms.

**Complexity results in context**

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

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

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

**Lessons.** Use theory as a guide.
- Ex. Don’t try to design sorting algorithm that guarantees $\frac{1}{2}N \log N$ compares.

**Complexity results in context (continued)**

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

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

**Duplicate keys.** Depending on the input distribution of duplicates, we may not need $N \log N$ 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: 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;
    }
}
```

Comparable interface: review

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);
...
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()`.

```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;
}
```

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;
        }
    }
}
```

Stability

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

```java
Selection.sort(a, Student.BY_NAME);
Selection.sort(a, Student.BY_SECTION);
```

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

```java
Arrays.sort(a, Student.BY_NAME);
Arrays.sort(a, Student.BY_SECTION);
```
Which sorts are stable?
A. Insertion sort and mergesort (but not selection sort or shellsort).

Note. Need to carefully check code ("less than" vs "less than or equal to").

Stability: insertion sort

Proposition. Insertion sort is stable.

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

Pf. Equal items never move past each other.

Stability: selection sort

Proposition. Selection sort is not stable.

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

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

Stability: shellsort

Proposition. Shellsort 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.
**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.

**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.