2.1 ELEMENTARY SORTS

- selection sort
- insertion sort
- sorting in Java
- comparators
- shuffling
- shell sort (if time permits, otherwise read textbook)
2.1 **Elementary Sorts**

- selection sort
- insertion sort
- sorting in Java
- comparators
- shuffling
- shell sort
Selection sort demo

- In iteration $i$, find index $\min$ of smallest remaining entry.
- Swap $a[i]$ and $a[\min]$.

*initial*
Selection sort demo

- In iteration $i$, find index $\min$ of smallest remaining entry.
- Swap $a[i]$ and $a[\min]$.
Selection sort demo

- In iteration $i$, find index $\min$ of smallest remaining entry.
- Swap $a[i]$ and $a[\min]$. 

![Diagram of selection sort with playing cards representing the elements and iteration process.](image)
Selection sort demo

- In iteration $i$, find index $\min$ of smallest remaining entry.
- Swap $a[i]$ and $a[\min]$.

Remaining entries
Selection sort demo

- In iteration $i$, find index $\min$ of smallest remaining entry.
- Swap $a[i]$ and $a[\min]$. 

in final order  
remaining entries
Selection sort demo

- In iteration $i$, find index $\text{min}$ of smallest remaining entry.
- Swap $a[i]$ and $a[\text{min}]$. 

![Selection sort demo](image)

in final order

remaining entries
Selection sort demo

- In iteration $i$, find index $\min$ of smallest remaining entry.
- Swap $a[i]$ and $a[\min]$. 

![Diagram of selection sort with playing cards](image)

- **in final order**
- **remaining entries**
Selection sort demo

- In iteration $i$, find index $\min$ of smallest remaining entry.
- Swap $a[i]$ and $a[\min]$.

![Diagram of selection sort demonstration with playing cards](image-url)
Selection sort demo

- In iteration $i$, find index $\min$ of smallest remaining entry.
- Swap $a[i]$ and $a[\min]$.
Selection sort demo

- In iteration $i$, find index $\min$ of smallest remaining entry.
- Swap $a[i]$ and $a[\min]$.
Selection sort demo

- In iteration $i$, find index $\min$ of smallest remaining entry.
- Swap $a[i]$ and $a[\min]$.

![Selection sort demonstration with cards](image)
Selection sort demo

- In iteration $i$, find index $\text{min}$ of smallest remaining entry.
- Swap $a[i]$ and $a[\text{min}]$. 

![Diagram showing selection sort process with playing cards.](diagram.png)

- **in final order**
- **remaining entries**
Selection sort demo

- In iteration $i$, find index $\min$ of smallest remaining entry.
- Swap $a[i]$ and $a[\min]$.

![Selection sort demo with cards](image)
Selection sort demo

- In iteration \( i \), find index \( \min \) of smallest remaining entry.
- Swap \( a[i] \) and \( a[\min] \).
Selection sort demo

- In iteration $i$, find index $\text{min}$ of smallest remaining entry.
- Swap $a[i]$ and $a[\text{min}]$. 

![Diagram showing selection sort process with cards in final order and remaining entries]
Selection sort demo

- In iteration $i$, find index $\min$ of smallest remaining entry.
- Swap $a[i]$ and $a[\min]$.

![Diagram of selection sort with cards representing the process.](image)
Selection sort demo

- In iteration $i$, find index $\min$ of smallest remaining entry.
- Swap $a[i]$ and $a[\min]$.

![Selection sort demo](image)
Selection sort demo

- In iteration $i$, find index $\min$ of smallest remaining entry.
- Swap $a[i]$ and $a[\min]$. 

in final order

remaining entries
Selection sort demo

- In iteration $i$, find index $\min$ of smallest remaining entry.
- Swap $a[i]$ and $a[\min]$. 

![Selection sort demo with cards]
Selection sort demo

- In iteration $i$, find index $\min$ of smallest remaining entry.
- Swap $a[i]$ and $a[\min]$.
Selection sort demo

- In iteration $i$, find index $\min$ of smallest remaining entry.
- Swap $a[i]$ and $a[\min]$. 

---

in final order

remaining entries
Selection sort demo

- In iteration $i$, find index $\min$ of smallest remaining entry.
- Swap $a[i]$ and $a[\min]$.

In final order

remaining entries
Selection sort demo

- In iteration $i$, find index $\min$ of smallest remaining entry.
- Swap $a[i]$ and $a[\min]$. 

![Selection sort diagram](image_url)
Selection sort demo

- In iteration $i$, find index $\min$ of smallest remaining entry.
- Swap $a[i]$ and $a[\min]$.
Selection sort demo

- In iteration $i$, find index $\min$ of smallest remaining entry.
- Swap $a[i]$ and $a[\min]$.

in final order

remaining entries
Selection sort demo

- In iteration $i$, find index $\min$ of smallest remaining entry.
- Swap $a[i]$ and $a[\min]$. 

![Selection sort diagram](image-url)
Selection sort demo

- In iteration $i$, find index $\min$ of smallest remaining entry.
- Swap $a[i]$ and $a[\min]$.
Selection sort demo

- In iteration $i$, find index $\min$ of smallest remaining entry.
- Swap $a[i]$ and $a[\min]$. 
Selection sort demo

- In iteration $i$, find index $\min$ of smallest remaining entry.
- Swap $a[i]$ and $a[\min]$.
Selection sort demo

- In iteration $i$, find index $\text{min}$ of smallest remaining entry.
- Swap $a[i]$ and $a[\text{min}]$.
Selection sort: mathematical analysis

Proposition. Selection sort uses \((N - 1) + (N - 2) + \ldots + 1 + 0 \sim N^2 / 2\) compares and \(N\) exchanges to sort any array of \(N\) items.

<table>
<thead>
<tr>
<th>(i)</th>
<th>(\text{min})</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>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a[\cdot])</td>
<td></td>
<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>(0)</td>
<td>6</td>
<td>\text{SORT} &amp; \text{EXAMP} &amp; \text{LE}</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>\text{SO} &amp; \text{RT} &amp; \text{E} &amp; \text{X} &amp; \text{AM} &amp; \text{PLE}</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>10</td>
<td>\text{AE} &amp; \text{RT} &amp; \text{E} &amp; \text{X} &amp; \text{S} &amp; \text{MP} &amp; \text{LE}</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>9</td>
<td>\text{AE} &amp; \text{E} &amp; \text{T} &amp; \text{O} &amp; \text{X} &amp; \text{S} &amp; \text{MP} &amp; \text{L} &amp; \text{R}</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>7</td>
<td>\text{AE} &amp; \text{EL} &amp; \text{O} &amp; \text{X} &amp; \text{S} &amp; \text{MP} &amp; \text{LE} &amp; \text{PTR}</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>7</td>
<td>\text{AE} &amp; \text{EL} &amp; \text{M} &amp; \text{X} &amp; \text{SO} &amp; \text{PTR}</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>8</td>
<td>\text{AE} &amp; \text{EL} &amp; \text{MO} &amp; \text{S} &amp; \text{X} &amp; \text{PTR}</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>10</td>
<td>\text{AE} &amp; \text{EL} &amp; \text{MO} &amp; \text{P} &amp; \text{X} &amp; \text{STR}</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>\text{AE} &amp; \text{EL} &amp; \text{MO} &amp; \text{PR} &amp; \text{S} &amp; \text{TX}</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td>\text{AE} &amp; \text{EL} &amp; \text{MO} &amp; \text{PR} &amp; \text{S} &amp; \text{TX}</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>\text{AE} &amp; \text{EL} &amp; \text{MO} &amp; \text{PR} &amp; \text{S} &amp; \text{TX}</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Entries in black are examined to find the minimum.
Entries in red are \(a[\text{min}]\).
Entries in gray are in final position.

Trace of selection sort (array contents just after each exchange)

Running time insensitive to input. Quadratic time, even if input is sorted.
Data movement is minimal. Linear number of exchanges—exactly \(N\).
2.1 Elementary Sorts

- selection sort
- insertion sort
- sorting in Java
- comparators
- shuffling
- shell sort
Insertion sort demo

- In iteration $i$, swap $a[i]$ with each larger entry to its left.
Insertion sort demo

- In iteration $i$, swap $a[i]$ with each larger entry to its left.
Insertion sort demo

- In iteration $i$, swap $a[i]$ with each larger entry to its left.
Insertion sort demo

- In iteration $i$, swap $a[i]$ with each larger entry to its left.
Insertion sort demo

- In iteration $i$, swap $a[i]$ with each larger entry to its left.
Insertion sort demo

- In iteration $i$, swap $a[i]$ with each larger entry to its left.
Insertion sort demo

- In iteration $i$, swap $a[i]$ with each larger entry to its left.
Insertion sort demo

- In iteration $i$, swap $a[i]$ with each larger entry to its left.
Insertion sort demo

- In iteration $i$, swap $a[i]$ with each larger entry to its left.
Insertion sort demo

- In iteration $i$, swap $a[i]$ with each larger entry to its left.
Insertion sort demo

- In iteration $i$, swap $a[i]$ with each larger entry to its left.
Insertion sort demo

- In iteration $i$, swap $a[i]$ with each larger entry to its left.
Insertion sort demo

- In iteration $i$, swap $a[i]$ with each larger entry to its left.
Insertion sort demo

- In iteration $i$, swap $a[i]$ with each larger entry to its left.
Insertion sort demo

- In iteration $i$, swap $a[i]$ with each larger entry to its left.
Insertion sort demo

- In iteration $i$, swap $a[i]$ with each larger entry to its left.
Insertion sort demo

- In iteration $i$, swap $a[i]$ with each larger entry to its left.
Insertion sort demo

- In iteration $i$, swap $a[i]$ with each larger entry to its left.
Insertion sort demo

- In iteration $i$, swap $a[i]$ with each larger entry to its left.
Insertion sort demo

- In iteration $i$, swap $a[i]$ with each larger entry to its left.
Insertion sort demo

- In iteration $i$, swap $a[i]$ with each larger entry to its left.
Insertion sort demo

- In iteration $i$, swap $a[i]$ with each larger entry to its left.
Insertion sort demo

- In iteration $i$, swap $a[i]$ with each larger entry to its left.

![Diagram showing insertion sort process with cards in ascending order and those not yet seen.](image)
Insertion sort demo

- In iteration $i$, swap $a[i]$ with each larger entry to its left.
Insertion sort demo

• In iteration $i$, swap $a[i]$ with each larger entry to its left.
Insertion sort demo

- In iteration $i$, swap $a[i]$ with each larger entry to its left.
Insertion sort demo

- In iteration $i$, swap $a[i]$ with each larger entry to its left.
Insertion sort demo

- In iteration $i$, swap $a[i]$ with each larger entry to its left.
Insertion sort demo

- In iteration $i$, swap $a[i]$ with each larger entry to its left.
Insertion sort demo

- In iteration $i$, swap $a[i]$ with each larger entry to its left.
Insertion sort demo

- In iteration $i$, swap $a[i]$ with each larger entry to its left.

[Diagram of cards in ascending order and not yet seen]
Insertion sort demo

- In iteration $i$, swap $a[i]$ with each larger entry to its left.

![Insertion sort demo with playing cards](image)
Insertion sort demo

- In iteration $i$, swap $a[i]$ with each larger entry to its left.
**Insertion sort demo**

- In iteration $i$, swap $a[i]$ with each larger entry to its left.
Insertion sort demo

- In iteration $i$, swap $a[i]$ with each larger entry to its left.
Insertion sort demo

- In iteration $i$, swap $a[i]$ with each larger entry to its left.
In iteration $i$, swap $a[i]$ with each larger entry to its left.
Insertion sort demo

- In iteration $i$, swap $a[i]$ with each larger entry to its left.
Insertion sort demo

- In iteration $i$, swap $a[i]$ with each larger entry to its left.
Insertion sort demo

- In iteration $i$, swap $a[i]$ with each larger entry to its left.
Insertion sort: animation

40 random items

http://www.sorting-algorithms.com/insertion-sort
### Insertion sort: mathematical analysis

**Proposition.** To sort a randomly-ordered array with distinct keys, insertion sort uses $\sim \frac{1}{4} N^2$ compares and $\sim \frac{1}{4} N^2$ exchanges on average.

**Pf.** Expect each entry to move halfway back.

<table>
<thead>
<tr>
<th>i</th>
<th>j</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>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>S</td>
<td>O</td>
<td>R</td>
<td>T</td>
<td>E</td>
<td>X</td>
<td>A</td>
<td>M</td>
<td>P</td>
<td>L</td>
<td>E</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>O</td>
<td>S</td>
<td>R</td>
<td>T</td>
<td>E</td>
<td>X</td>
<td>A</td>
<td>M</td>
<td>P</td>
<td>L</td>
<td>E</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>O</td>
<td>R</td>
<td>S</td>
<td>T</td>
<td>E</td>
<td>X</td>
<td>A</td>
<td>M</td>
<td>P</td>
<td>L</td>
<td>E</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>O</td>
<td>R</td>
<td>S</td>
<td>T</td>
<td>E</td>
<td>X</td>
<td>A</td>
<td>M</td>
<td>P</td>
<td>L</td>
<td>E</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>E</td>
<td>O</td>
<td>R</td>
<td>S</td>
<td>T</td>
<td>E</td>
<td>X</td>
<td>A</td>
<td>M</td>
<td>P</td>
<td>L</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>E</td>
<td>O</td>
<td>R</td>
<td>S</td>
<td>T</td>
<td>X</td>
<td>A</td>
<td>M</td>
<td>P</td>
<td>L</td>
<td>E</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>A</td>
<td>E</td>
<td>O</td>
<td>R</td>
<td>S</td>
<td>T</td>
<td>X</td>
<td>A</td>
<td>M</td>
<td>P</td>
<td>L</td>
</tr>
<tr>
<td>7</td>
<td>2</td>
<td>A</td>
<td>E</td>
<td>M</td>
<td>O</td>
<td>R</td>
<td>S</td>
<td>T</td>
<td>X</td>
<td>P</td>
<td>L</td>
<td>E</td>
</tr>
<tr>
<td>8</td>
<td>4</td>
<td>A</td>
<td>E</td>
<td>M</td>
<td>O</td>
<td>P</td>
<td>R</td>
<td>S</td>
<td>T</td>
<td>X</td>
<td>L</td>
<td>E</td>
</tr>
<tr>
<td>9</td>
<td>2</td>
<td>A</td>
<td>E</td>
<td>L</td>
<td>M</td>
<td>O</td>
<td>P</td>
<td>R</td>
<td>S</td>
<td>T</td>
<td>X</td>
<td>E</td>
</tr>
<tr>
<td>10</td>
<td>2</td>
<td>A</td>
<td>E</td>
<td>E</td>
<td>L</td>
<td>M</td>
<td>O</td>
<td>P</td>
<td>R</td>
<td>S</td>
<td>T</td>
<td>X</td>
</tr>
</tbody>
</table>

Trace of insertion sort (array contents just after each insertion)
Insertion sort: animation

40 reverse-sorted items

http://www.sorting-algorithms.com/insertion-sort
Insertion sort: analysis

**Worst case.** If the array is in descending order (and no duplicates), insertion sort makes $\sim \frac{1}{2} N^2$ compares and $\sim \frac{1}{2} N^2$ exchanges.

```
X T S R P O M L F E A
```

**Best case.** If the array is in ascending order, insertion sort makes $N-1$ compares and 0 exchanges.

```
A E E L M O P R S T X
```
Insertion sort: animation

40 partially-sorted items

http://www.sorting-algorithms.com/insertion-sort
Selection sort: animations

20 partially-sorted items

http://www.sorting-algorithms.com/selection-sort
Insertion sort: partially-sorted arrays

**Def.** An inversion is a pair of keys that are out of order.

\[
\begin{array}{cccccccccccc}
A & E & E & L & M & O & T & R & X & P & S \\
\end{array}
\]

- \(T-R\) \(T-P\) \(T-S\) \(R-P\) \(X-P\) \(X-S\)

(6 inversions)

**Def.** An array is **partially sorted** if the number of inversions is \(\leq cN\).

- Ex 1. A sorted array has 0 inversions.
- Ex 2. A subarray of size 10 appended to a sorted subarray of size \(N\).

**Proposition.** For partially-sorted arrays, insertion sort runs in linear time.

**Pf.** Number of exchanges equals the number of inversions.

\[
\text{number of compares} \leq \text{exchanges} + (N - 1)
\]
Insertion sort demo

- In iteration $i$, swap $a[i]$ with each larger entry to its left.

https://www.youtube.com/watch?v=ROalU379I3U
2.1 Elementary Sorts

- selection sort
- insertion sort
- sorting in Java
- comparators
- shuffling
Sample sort client 1

Goal. Sort any type of data.

Ex 1. Sort random real numbers in ascending order.

seems artificial (stay tuned for an application)

```java
public class Experiment {
    public static void main(String[] args) {
        int N = Integer.parseInt(args[0]);
        Double[] a = new Double[N];
        for (int i = 0; i < N; i++)
            a[i] = StdRandom.uniform();
        Insertion.sort(a);
        for (int i = 0; i < N; i++)
            StdOut.println(a[i]);
    }
}
```

% java Experiment 10
0.08614716385210452
0.09054270895414829
0.10708746304898642
0.21166190071646818
0.363292849257276
0.460954145685913
0.5340026311350087
0.7216129793703496
0.9003500354411443
0.9293994908845686
Sample sort client 2

**Goal.** Sort any type of data.

**Ex 2.** Sort strings in alphabetical order.

```java
public class StringSorter {
    public static void main(String[] args) {
        String[] a = StdIn.readAllStrings();
        Insertion.sort(a);
        for (int i = 0; i < a.length; i++)
            StdOut.println(a[i]);
    }
}

% more words3.txt
bed bug dad yet zoo ... all bad yes

% java StringSorter < words3.txt
all bad bed bug dad ... yes yet zoo [suppressing newlines]
```
Sample sort client 3

**Goal.** Sort any type of data.

**Ex 3.** Sort the files in a given directory by filename.

```java
import java.io.File;

public class FileSorter {
    public static void main(String[] args) {
        File directory = new File(args[0]);
        File[] files = directory.listFiles();
        Insertion.sort(files);
        for (int i = 0; i < files.length; i++)
            StdOut.println(files[i].getName());
    }
}
```
Callbacks

**Goal.** Sort any type of data (for which sorting is well defined).

**Q.** How can `sort()` compare data of type `Double`, `String`, and `java.io.File` without hardwiring in type-specific information.

Callback = reference to executable code.
- Client passes array of objects to `sort()` function.
- The `sort()` function calls object's `compareTo()` method as needed.

Implementing callbacks.
- **Java:** interfaces.
- **C:** function pointers.
- **C++:** class-type functors.
- **C#:** delegates.
- **Python, Perl, ML, Javascript:** first-class functions.
Callbacks: Java interfaces

Interface. Specifies a set of methods that a concrete class can provide.

```
public interface Comparable<Item> {
    public int compareTo(Item that);
}
```

Concrete class. Can provide the set of methods in the interface.

```
public class String implements Comparable<String> {
    ...
    public int compareTo(String that) {
        ...
    }
}
```

Impact.
- You can treat any `String` object as an object of type `Comparable`.
- On a `Comparable` object, you can invoke (only) the `compareTo()` method.
- Enables callbacks.
Callbacks: roadmap

client (StringSorter.java)

```java
public class StringSorter {
    public static void main(String[] args) {
        String[] a = StdIn.readAllStrings();
        Insertion.sort(a);
        for (int i = 0; i < a.length; i++)
            StdOut.println(a[i]);
    }
}
```

java.lang.Comparable interface

```java
public interface Comparable<Item> {
    public int compareTo(Item that);
}
```

data type implementation (String.java)

```java
public class String implements Comparable<String> {
    ... 
    public int compareTo(String that) {
        ... 
    }
}
```

sort implementation (Insertion.java)

```java
public static void sort(Comparable[] a) {
    int N = a.length;
    for (int i = 0; i < N; i++)
        for (int j = i; j > 0; j--)
            if (a[j].compareTo(a[j-1]) < 0)
                exch(a, j, j-1);
        else break;
}
```

key point: no dependence on type of data to be sorted
java.lang.Comparable API

Implement `compareTo()` so that `v.compareTo(w)`

- Defines a total order.
- Returns a negative integer, zero, or positive integer if `v` is less than, equal to, or greater than `w`, respectively.
- Throws an exception if incompatible types (or either is `null`).

![Scale comparisons](image)

**Less than** (return negative integer)

**Equal to** (return 0)

**Greater than** (return positive integer)

Built-in comparable types. Integer, Double, String, Date, File, ...

User-defined comparable types. Implement the Comparable interface.
Implementing the Comparable interface

**Date data type.** Simplified version of `java.util.Date`.

```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;
    }
}
```

Implementing the `Comparable` interface can compare `Date` objects only to other `Date` objects.

[http://algs4.cs.princeton.edu/12oop/Date.java.html](http://algs4.cs.princeton.edu/12oop/Date.java.html)
Two useful sorting abstractions

Helper functions. Refer to data only through compares and exchanges.

Less. Is item \( v \) less than \( w \)?

```java
private static boolean less(Comparable v, Comparable w) {
    return v.compareTo(w) < 0;
}
```

Exchange. Swap item in array \( a[] \) at index \( i \) with the one at index \( j \).

```java
private static void exch(Object[] a, int i, int j) {
    Object swap = a[i];
    a[i] = a[j];
    a[j] = swap;
}
```
Total order

Goal. Sort any type of data (for which sorting is well defined).

A total order is a binary relation $\leq$ that satisfies:

- **Antisymmetry:** if both $v \leq w$ and $w \leq v$, then $v = w$.
- **Transitivity:** if both $v \leq w$ and $w \leq x$, then $v \leq x$.
- **Totality:** either $v \leq w$ or $w \leq v$ or both.

Ex.

- Standard order for natural and real numbers.
- Chronological order for dates or times.
- Lexicographic order for strings.
2.1 Elementary Sorts

- selection sort
- insertion sort
- sorting in Java
- comparators
- shuffling
Sort music library by artist
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;
    }
}
```
Comparator interface

Comparator interface: sort using an alternate order.

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

Required property. Must be a total order.

<table>
<thead>
<tr>
<th>string order</th>
<th>example</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>natural order</td>
<td>Now is the time</td>
<td></td>
</tr>
<tr>
<td>case insensitive</td>
<td>is Now the time</td>
<td>pre-1994 order for</td>
</tr>
<tr>
<td></td>
<td></td>
<td>digraphs ch and ll and rr</td>
</tr>
<tr>
<td>Spanish language</td>
<td>café cafetero cuarto churro nube ñoño</td>
<td></td>
</tr>
<tr>
<td>British phone book</td>
<td>McKinley Mackintosh</td>
<td></td>
</tr>
</tbody>
</table>
Comparator interface: system sort

To use with Java system sort:

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

```
String[] a;
...
Arrays.sort(a);
...
Arrays.sort(a, String.CASE_INSENSITIVE_ORDER);
...
Arrays.sort(a, Collator.getInstance(new Locale("es")));
...
Arrays.sort(a, new BritishPhoneBookOrder());
...
```

Bottom line. Decouples the definition of the data type from the definition of what it means to compare two objects of that type.
Comparator interface: using with our sorting libraries

To support comparators in our sort implementations:

- Pass Comparator to both sort() and less(), and use it in less().
- Use Object instead of Comparable.

```java
import java.util.Comparator;

public class Insertion {
    ...

    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 comparator, Object v, Object w) {
        return comparator.compare(v, w) < 0;
    }
}
```

http://algs4.cs.princeton.edu/21elementary/Insertion.java.html
Comparator interface: implementing

To implement a comparator:
- Define a (nested) class that implements the Comparator interface.
- Implement the compare() method.
- Provide client access to Comparator.

```java
import java.util.Comparator;

public class Student {
    private final String name;
    private final int section;
    ...

    public static Comparator<Student> nameOrder() {
        return new NameOrder();
    }

    private static class NameOrder implements Comparator<Student> {
        public int compare(Student v, Student w) {
            return v.name.compareTo(w.name);
        }
    }

    ...
}
```
Comparator interface: implementing

To implement a comparator:

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

```java
import java.util.Comparator;

public class Student
{
    private final String name;
    private final int section;
    ...

    public static Comparator<Student> sectionOrder()
    { return new SectionOrder(); }

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

    ...
}
```

this trick works here since no danger of overflow
Comparator interface: implementing

To implement a comparator:

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

```java
Comparator interface: implementing

To implement a comparator:

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

```
Stability

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

```
Selection.sort(a, Student.nameOrder());
```

<table>
<thead>
<tr>
<th>Name</th>
<th>Section</th>
<th>Area</th>
<th>Phone</th>
<th>Room</th>
</tr>
</thead>
<tbody>
<tr>
<td>Andrews</td>
<td>3</td>
<td>A</td>
<td>(664) 480-0023</td>
<td>097 Little</td>
</tr>
<tr>
<td>Battle</td>
<td>4</td>
<td>C</td>
<td>(874) 088-1212</td>
<td>121 Whitman</td>
</tr>
<tr>
<td>Chen</td>
<td>3</td>
<td>A</td>
<td>(991) 878-4944</td>
<td>308 Blair</td>
</tr>
<tr>
<td>Fox</td>
<td>3</td>
<td>A</td>
<td>(884) 232-5341</td>
<td>11 Dickinson</td>
</tr>
<tr>
<td>Furia</td>
<td>1</td>
<td>A</td>
<td>(766) 093-9873</td>
<td>101 Brown</td>
</tr>
<tr>
<td>Gazsi</td>
<td>4</td>
<td>B</td>
<td>(800) 867-5309</td>
<td>101 Brown</td>
</tr>
<tr>
<td>Kanaga</td>
<td>3</td>
<td>B</td>
<td>(898) 122-9643</td>
<td>22 Brown</td>
</tr>
<tr>
<td>Rohde</td>
<td>2</td>
<td>A</td>
<td>(232) 343-5555</td>
<td>343 Forbes</td>
</tr>
</tbody>
</table>

```
Selection.sort(a, Student.sectionOrder());
```

<table>
<thead>
<tr>
<th>Name</th>
<th>Section</th>
<th>Area</th>
<th>Phone</th>
<th>Room</th>
</tr>
</thead>
<tbody>
<tr>
<td>Furia</td>
<td>1</td>
<td>A</td>
<td>(766) 093-9873</td>
<td>101 Brown</td>
</tr>
<tr>
<td>Rohde</td>
<td>2</td>
<td>A</td>
<td>(232) 343-5555</td>
<td>343 Forbes</td>
</tr>
<tr>
<td>Chen</td>
<td>3</td>
<td>A</td>
<td>(991) 878-4944</td>
<td>308 Blair</td>
</tr>
<tr>
<td>Fox</td>
<td>3</td>
<td>A</td>
<td>(884) 232-5341</td>
<td>11 Dickinson</td>
</tr>
<tr>
<td>Andrews</td>
<td>3</td>
<td>A</td>
<td>(664) 480-0023</td>
<td>097 Little</td>
</tr>
<tr>
<td>Kanaga</td>
<td>3</td>
<td>B</td>
<td>(898) 122-9643</td>
<td>22 Brown</td>
</tr>
<tr>
<td>Gazsi</td>
<td>4</td>
<td>B</td>
<td>(800) 867-5309</td>
<td>101 Brown</td>
</tr>
<tr>
<td>Battle</td>
<td>4</td>
<td>C</td>
<td>(874) 088-1212</td>
<td>121 Whitman</td>
</tr>
</tbody>
</table>

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

A stable sort preserves the relative order of items with equal keys.
Stability: 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);
    }
}
```

<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.

```java
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);
        }
    }
}
```

| $i$ | $\text{min}$ | $0$ | $1$ | $2$
|-----|-------------|-----|-----|-----
| 0   | 2           | $B_1$ | $B_2$ | $A$
| 1   | 1           | $A$ | $B_2$ | $B_1$
| 2   | 2           | $A$ | $B_2$ | $B_1$

Pf by counterexample. Long-distance exchange can move one equal item past another one.
2.1 **Elementary Sorts**

- selection sort
- insertion sort
- sorting in Java
- comparators
- shuffling
Shuffle sort

- Generate a random real number for each array entry.
- Sort the array.

\[
\begin{array}{cccccccc}
2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\
\clubsuit & \clubsuit & \clubsuit & \clubsuit & \clubsuit & \clubsuit & \clubsuit & \clubsuit & \clubsuit \\
0.8003 & 0.9706 & 0.9157 & 0.9649 & 0.1576 & 0.4854 & 0.1419 & 0.4218 & 0.9572 \\
\end{array}
\]
Shuffle sort

- Generate a random real number for each array entry.
- Sort the array.
Shuffle sort

- Generate a random real number for each array entry.
- Sort the array.

**Proposition.** Shuffle sort produces a uniformly random permutation.

**Application.** Shuffle columns in a spreadsheet.

assuming real numbers are uniformly random (and no ties)
War story (Microsoft)

Microsoft antitrust probe by EU. Microsoft agreed to provide a randomized ballot screen for users to select browser in Windows 7.

http://www.browserchoice.eu

Select your web browser(s)

- Google Chrome: A fast new browser from Google. Try it now!
- Safari: Safari for Windows from Apple, the world’s most innovative browser.
- Mozilla Firefox: Your online security is Firefox’s top priority. Firefox is free, and made to help you get the most out of the Web.
- Opera: The fastest browser on Earth. Secure, powerful and easy to use, with excellent privacy protection.
- Internet Explorer 8: Designed to help you take control of your privacy and browse with confidence. Free from Microsoft.

Appeared last 50% of the time
War story (Microsoft)

Microsoft antitrust probe by EU. Microsoft agreed to provide a randomized ballot screen for users to select browser in Windows 7.

Solution? Implement shuffle sort by making comparator always return a random answer.

```java
public int compareTo(Browser that)
{
    double r = Math.random();
    if (r < 0.5) return -1;
    if (r > 0.5) return +1;
    return 0;
}
```
Knuth shuffle demo

- In iteration $i$, pick integer $r$ between 0 and $i$ uniformly at random.
- Swap $a[i]$ and $a[r]$. 
Knuth shuffle

- In iteration $i$, pick integer $r$ between 0 and $i$ uniformly at random.
- Swap $a[i]$ and $a[r]$.

![Diagram of the Knuth shuffle with playing cards](image-url)
Knuth shuffle

- In iteration $i$, pick integer $r$ between 0 and $i$ uniformly at random.
- Swap $a[i]$ and $a[r]$. 

![Card images showing the Knuth shuffle process](image)
Knuth shuffle

- In iteration $i$, pick integer $r$ between 0 and $i$ uniformly at random.
- Swap $a[i]$ and $a[r]$.
Knuth shuffle

- In iteration $i$, pick integer $r$ between 0 and $i$ uniformly at random.
- Swap $a[i]$ and $a[r]$.

![Diagram of Knuth shuffle process with cards](image_url)
Knuth shuffle

- In iteration \( i \), pick integer \( r \) between 0 and \( i \) uniformly at random.
- Swap \( a[i] \) and \( a[r] \).
Knuth shuffle

- In iteration $i$, pick integer $r$ between 0 and $i$ uniformly at random.
- Swap $a[i]$ and $a[r]$.
Knuth shuffle

- In iteration $i$, pick integer $r$ between 0 and $i$ uniformly at random.
- Swap $a[i]$ and $a[r]$.
Knuth shuffle

- In iteration \( i \), pick integer \( r \) between 0 and \( i \) uniformly at random.
- Swap \( a[i] \) and \( a[r] \).

\[ r \]
\[ i \]

4♠ 2♣ 3♣ 5 6 7 8 9 10

shuffled

not yet seen
Knuth shuffle

- In iteration $i$, pick integer $r$ between 0 and $i$ uniformly at random.
- Swap $a[i]$ and $a[r]$.

![Diagram of the Knuth shuffle process]

- Shuffled cards: 4, 2, 3
- Not yet seen cards: 5, 6, 7, 8, 9, 10
Knuth shuffle

- In iteration $i$, pick integer $r$ between 0 and $i$ uniformly at random.
- Swap $a[i]$ and $a[r]$.
Knuth shuffle

- In iteration $i$, pick integer $r$ between 0 and $i$ uniformly at random.
- Swap $a[i]$ and $a[r]$.

![Card deck example]
Knuth shuffle

- In iteration $i$, pick integer $r$ between 0 and $i$ uniformly at random.
- Swap $a[i]$ and $a[r]$.
Knuth shuffle

- In iteration $i$, pick integer $r$ between 0 and $i$ uniformly at random.
- Swap $a[i]$ and $a[r]$.
Knuth shuffle

- In iteration \( i \), pick integer \( r \) between 0 and \( i \) uniformly at random.
- Swap \( a[i] \) and \( a[r] \).
Knuth shuffle

- In iteration $i$, pick integer $r$ between 0 and $i$ uniformly at random.
- Swap $a[i]$ and $a[r]$.
Knuth shuffle

- In iteration $i$, pick integer $r$ between 0 and $i$ uniformly at random.
- Swap $a[i]$ and $a[r]$.
Knuth shuffle

- In iteration $i$, pick integer $r$ between 0 and $i$ uniformly at random.
- Swap $a[i]$ and $a[r]$. 

![Diagram of the Knuth shuffle process](attachment:knuth_shuffle_diagram.png)
Knuth shuffle

- In iteration $i$, pick integer $r$ between 0 and $i$ uniformly at random.
- Swap $a[i]$ and $a[r]$. 

![Diagram showing a Knuth shuffle example with cards and numbers indicating shuffled and not yet seen areas.](image-url)
Knuth shuffle

- In iteration $i$, pick integer $r$ between 0 and $i$ uniformly at random.
- Swap $a[i]$ and $a[r]$.
Knuth shuffle

- In iteration \( i \), pick integer \( r \) between 0 and \( i \) uniformly at random.
- Swap \( a[i] \) and \( a[r] \).
Knuth shuffle

- In iteration $i$, pick integer $r$ between 0 and $i$ uniformly at random.
- Swap $a[i]$ and $a[r]$.
Knuth shuffle

- In iteration $i$, pick integer $r$ between 0 and $i$ uniformly at random.
- Swap $a[i]$ and $a[r]$.
Knuth shuffle

- In iteration $i$, pick integer $r$ between 0 and $i$ uniformly at random.
- Swap $a[i]$ and $a[r]$.
Knuth shuffle

- In iteration $i$, pick integer $r$ between 0 and $i$ uniformly at random.
- Swap $a[i]$ and $a[r]$.

![Card deck diagram]
Knuth shuffle

- In iteration $i$, pick integer $r$ between 0 and $i$ uniformly at random.
- Swap $a[i]$ and $a[r]$. 

Diagram:

```
  7 6 5 3 8 4 2 9 10
```

shuffled
Knuth shuffle

- In iteration $i$, pick integer $r$ between 0 and $i$ uniformly at random.
- Swap $a[i]$ and $a[r]$.
Knuth shuffle

- In iteration $i$, pick integer $r$ between 0 and $i$ uniformly at random.
- Swap $a[i]$ and $a[r]$.
Knuth shuffle

- In iteration $i$, pick integer $r$ between 0 and $i$ uniformly at random.
- Swap $a[i]$ and $a[r]$.

**Proposition.** [Fisher-Yates 1938] Knuth shuffling algorithm produces a uniformly random permutation of the input array in linear time.
Knuth shuffle

- In iteration $i$, pick integer $r$ between 0 and $i$ uniformly at random.
- Swap $a[i]$ and $a[r]$.

```java
public class Knuth {
    public static void shuffle(Object[] a) {
        int N = a.length;
        for (int i = 0; i < N; i++) {
            int r = StdRandom.uniform(i + 1);
            exch(a, i, r);
        }
    }
}
```

http://algs4.cs.princeton.edu/11model/Knuth.java.html
Broken Knuth shuffle

**Q.** What happens if integer is chosen between 0 and \( N-1 \) ?

**A.** Not uniformly random!

<table>
<thead>
<tr>
<th>permutation</th>
<th>Knuth shuffle</th>
<th>broken shuffle</th>
</tr>
</thead>
<tbody>
<tr>
<td>A B C</td>
<td>$1/6$</td>
<td>$4/27$</td>
</tr>
<tr>
<td>A C B</td>
<td>$1/6$</td>
<td>$5/27$</td>
</tr>
<tr>
<td>B A C</td>
<td>$1/6$</td>
<td>$5/27$</td>
</tr>
<tr>
<td>B C A</td>
<td>$1/6$</td>
<td>$5/27$</td>
</tr>
<tr>
<td>C A B</td>
<td>$1/6$</td>
<td>$4/27$</td>
</tr>
<tr>
<td>C B A</td>
<td>$1/6$</td>
<td>$4/27$</td>
</tr>
</tbody>
</table>

probability of each permutation when shuffling \{ A, B, C \}
War story (online poker)

**Texas hold'em poker.** Software must shuffle electronic cards.

How We Learned to Cheat at Online Poker: A Study in Software Security

War story (online poker)

Shuffling algorithm in FAQ at www.planetpoker.com

```
for i := 1 to 52 do begin
    r := random(51) + 1;
    swap := card[r];
    card[r] := card[i];
    card[i] := swap;
end;
```

**Bug 1.** Random number \( r \) never 52 \( \Rightarrow \) 52\textsuperscript{nd} card can't end up in 52\textsuperscript{nd} place.

**Bug 2.** Shuffle not uniform (should be between 1 and i).

**Bug 3.** \texttt{random()} uses 32-bit seed \( \Rightarrow \) \(2^{32}\) possible shuffles.

**Bug 4.** Seed = milliseconds since midnight \( \Rightarrow \) 86.4 million shuffles.

“The generation of random numbers is too important to be left to chance.”

— Robert R. Coveyou
War story (online poker)

Best practices for shuffling (if your business depends on it).

- Use a hardware random-number generator that has passed both the FIPS 140-2 and the NIST statistical test suites.
- Continuously monitor statistic properties: hardware random-number generators are fragile and fail silently.
- Use an unbiased shuffling algorithm.

Bottom line. Shuffling a deck of cards is hard!
2.1 Elementary Sorts

- rules of the game
- selection sort
- insertion sort
- shuffling
- comparators
- shellsort
ShellSort overview

**Idea.** Move entries more than one position at a time by \( h \)-\textit{sorting} the array.

\[
\text{an } h \text{-sorted array is } h \text{ interleaved sorted subsequences}
\]

\[
h = 4 \\
\begin{array}{cccccccccccccc}
L & E & E & A & M & H & L & E & P & S & O & L & T & S & X & R \\
L & U & U & U & M & U & U & P & U & T \\
E & U & U & H & S & U & U & S \\
E & U & U & L & O & U & U & X \\
A & U & U & E & U & U & L & R
\end{array}
\]

**ShellSort.** [Shell 1959] \( h \)-\textit{sort} array for decreasing sequence of values of \( h \).

<table>
<thead>
<tr>
<th>input</th>
<th>SHELL SORT EXAM PLE</th>
</tr>
</thead>
<tbody>
<tr>
<td>13-sort</td>
<td>PHELL SORT EXAM SLE</td>
</tr>
<tr>
<td>4-sort</td>
<td>LEEAMHLEPSOLT SXR</td>
</tr>
<tr>
<td>1-sort</td>
<td>AEEEHLLLMOPRSSTX</td>
</tr>
</tbody>
</table>
h-sorting demo

In iteration $i$, swap $a[i]$ with each larger entry $h$ positions to its left.
h-sorting

How to $h$-sort an array? Insertion sort, with stride length $h$.

3-sorting an array

```
M O L E E X A S P R T
E O L M E X A S P R T
E E L M O X A S P R T
E E L M O X A S P R T
A E L E O X M S P R T
A E L E O X M S P R T
A E L E O P M S X R T
A E L E O P M S X R T
A E L E O P M S X R T
A E L E O P M S X R T
```

Why insertion sort?

- Big increments ⇒ small subarray.
- Small increments ⇒ nearly in order. [stay tuned]
Shell sort example: increments 7, 3, 1

**input**

```
SORTEXAMPLE
```

**7-sort**

```
SORTEXAMPLE
MORTEXAMPLE
MORTEXASPLE
MOLTEXASPR
MOLEXASPR
```

**3-sort**

```
MOLEXASPR
EOLMEXASPR
EELOXASPR
EELOXASPR
AELEOXMSPR
AELEOXMSPR
AELEOPMSPR
AELEOPMSXR
AELEOPMSXR
```

**1-sort**

```
AELEOPMSXRT
AELEOPMSXRT
AELEOPMSXRT
AELELOPMSXRT
AELELOPMSXRT
AELELOPMSXRT
AELELOPMSXRT
AELELOPMSXRT
AELELOPMSXRT
```

**result**

```
AEELMOPRSTX
```

151
**Shellsort: Java implementation**

```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; // 1, 4, 13, 40, 121, 364, ...

        while (h >= 1)
        { // h-sort the array.
            for (int i = h; i < N; i++)
            {
                for (int j = i; j >= h && less(a[j], a[i-h]); j -= h)
                    exch(a, j, j-h);
            }
            h = h/3;
        }
    }
}

private static boolean less(Comparable v, Comparable w)
{ /* as before */ }
private static void exch(Comparable[] a, int i, int j)
{ /* as before */ }
```

Shellsort: visual trace

input

40-sorted

13-sorted

4-sorted

result
Shellsort: animation

50 random items

http://www.sorting-algorithms.com/shell-sort
Shell sort: animation

50 partially-sorted items

http://www.sorting-algorithms.com/shell-sort
Shellsort: which increment sequence to use?

Powers of two. 1, 2, 4, 8, 16, 32, ...
No.

Powers of two minus one. 1, 3, 7, 15, 31, 63, ...
Maybe.

→ 3x + 1. 1, 4, 13, 40, 121, 364, ...
OK. Easy to compute.

Sedgewick. 1, 5, 19, 41, 109, 209, 505, 929, 2161, 3905, ...
Good. Tough to beat in empirical studies.

merging of \((9 \times 4^i) - (9 \times 2^i) + 1\)
and \(4^i - (3 \times 2^i) + 1\)
Proposition. An \( h \)-sorted array remains \( h \)-sorted after \( g \)-sorting it.

Challenge. Prove this fact—it's more subtle than you'd think!
Proposition. The order of growth of the worst-case number of compares used by shellsort with the $3x+1$ increments is $N^{3/2}$.

Property. The expected number of compares to shellsort a randomly-ordered array using $3x+1$ increments is....

<table>
<thead>
<tr>
<th>N</th>
<th>compares</th>
<th>$2.5 \times N \ln N$</th>
<th>$0.25 \times N \ln^2 N$</th>
<th>$N^{1.3}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>5,000</td>
<td>93K</td>
<td>106K</td>
<td>91K</td>
<td>64K</td>
</tr>
<tr>
<td>10,000</td>
<td>209K</td>
<td>230K</td>
<td>213K</td>
<td>158K</td>
</tr>
<tr>
<td>20,000</td>
<td>467K</td>
<td>495K</td>
<td>490K</td>
<td>390K</td>
</tr>
<tr>
<td>40,000</td>
<td>1022K</td>
<td>1059K</td>
<td>1122K</td>
<td>960K</td>
</tr>
<tr>
<td>80,000</td>
<td>2266K</td>
<td>2258K</td>
<td>2549K</td>
<td>2366K</td>
</tr>
</tbody>
</table>

Remark. Accurate model has not yet been discovered (!)
Why are we interested in shellsort?

Example of simple idea leading to substantial performance gains.

Useful in practice.
- Fast unless array size is huge (used for small subarrays).
- Tiny, fixed footprint for code (used in some embedded systems).
- Hardware sort prototype.

Simple algorithm, nontrivial performance, interesting questions.
- Asymptotic growth rate?
- Best sequence of increments?
- Average-case performance?

Lesson. Some good algorithms are still waiting discovery.
**Elementary sorts summary**

**Today.** Elementary sorting algorithms.

<table>
<thead>
<tr>
<th>algorithm</th>
<th>best</th>
<th>average</th>
<th>worst</th>
</tr>
</thead>
<tbody>
<tr>
<td>selection sort</td>
<td>$N^2$</td>
<td>$N^2$</td>
<td>$N^2$</td>
</tr>
<tr>
<td>insertion sort</td>
<td>$N$</td>
<td>$N^2$</td>
<td>$N^2$</td>
</tr>
<tr>
<td>Shellsort (3x+1)</td>
<td>$N \log N$</td>
<td>?</td>
<td>$N^{3/2}$</td>
</tr>
<tr>
<td>goal</td>
<td>$N$</td>
<td>$N \log N$</td>
<td>$N \log N$</td>
</tr>
</tbody>
</table>

*order of growth of running time to sort an array of $N$ items*

**Next week.** $N \log N$ sorting algorithms (in worst case).