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

- Sorting review
- Rules of the game
- Selection sort
- Insertion sort
- Shellsort
ELEMENTARY SORTING ALGORITHMS

- Sorting review
- Rules of the game
- Selection sort
- Insertion sort
- Shellsort
**Ex.** Student records in a university.

<table>
<thead>
<tr>
<th>Item</th>
<th>Key</th>
<th>Value 1</th>
<th>Value 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chen</td>
<td>3</td>
<td>991-878-4944</td>
<td>308 Blair</td>
</tr>
<tr>
<td>Rohde</td>
<td>2</td>
<td>232-343-5555</td>
<td>343 Forbes</td>
</tr>
<tr>
<td>Gazsi</td>
<td>4</td>
<td>766-093-9873</td>
<td>101 Brown</td>
</tr>
<tr>
<td>Furia</td>
<td>1</td>
<td>766-093-9873</td>
<td>101 Brown</td>
</tr>
<tr>
<td>Kanaga</td>
<td>3</td>
<td>898-122-9643</td>
<td>22 Brown</td>
</tr>
<tr>
<td>Andrews</td>
<td>3</td>
<td>664-480-0023</td>
<td>097 Little</td>
</tr>
<tr>
<td>Battle</td>
<td>4</td>
<td>874-088-1212</td>
<td>121 Whitman</td>
</tr>
</tbody>
</table>

**Sort.** Rearrange array of $N$ items into ascending order.

<table>
<thead>
<tr>
<th>Item</th>
<th>Key</th>
<th>Value 1</th>
<th>Value 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Andrews</td>
<td>3</td>
<td>664-480-0023</td>
<td>097 Little</td>
</tr>
<tr>
<td>Battle</td>
<td>4</td>
<td>874-088-1212</td>
<td>121 Whitman</td>
</tr>
<tr>
<td>Chen</td>
<td>3</td>
<td>991-878-4944</td>
<td>308 Blair</td>
</tr>
<tr>
<td>Furia</td>
<td>1</td>
<td>766-093-9873</td>
<td>101 Brown</td>
</tr>
<tr>
<td>Gazsi</td>
<td>4</td>
<td>766-093-9873</td>
<td>101 Brown</td>
</tr>
<tr>
<td>Kanaga</td>
<td>3</td>
<td>898-122-9643</td>
<td>22 Brown</td>
</tr>
<tr>
<td>Rohde</td>
<td>2</td>
<td>232-343-5555</td>
<td>343 Forbes</td>
</tr>
</tbody>
</table>
**Goal.** Sort *any* type of data.

**Ex 1.** Sort random real numbers in ascending order.

Sample sort client

```
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

seems artificial, but stay tuned for an application
Goal. Sort any type of data.

Ex 2. Sort strings from file in alphabetical order.

```java
public class StringSorter
{
    public static void main(String[] args)
    {
        String[] a = In.readStrings(args[0]);
        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
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.

**Q.** How can `sort()` know how to compare data of type `Double`, `String`, and `java.io.File` without any information about the type of an item's key?

**Callback = reference to executable code.**
- Client passes array of objects to `sort()` function.
- The `sort()` function calls back 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.
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());
    }
}

Comparable interface (built in to Java)

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

object implementation

public class File
implements Comparable<File>
{
...
public int compareTo(File b)
{
...
    return -1;
...
    return +1;
...
    return 0;
}
}

sort implementation

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;
}
A **total order** is a binary relation $\leq$ that satisfies
- **Antisymmetry:** if $v \leq w$ and $w \leq v$, then $v = w$.
- **Transitivity:** if $v \leq w$ 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.
- Alphabetical order for strings.
- Chronological order for dates.
- ...
Comparable API

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

- Is 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).

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

User-defined comparable types. Implement 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 only compare dates to other dates.
Two useful sorting abstractions

**Helper functions.** Refer to data 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(Comparable[] a, int i, int j)
{
    Comparable swap = a[i];
    a[i] = a[j];
    a[j] = swap;
}
```
Elementary Sorting Algorithms

- Sorting review
- Rules of the game
- Selection sort
- Insertion sort
- Shellsort
Selection sort

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

![Remaining entries image]
In iteration $i$, find index $\min$ of smallest remaining entry.

Swap $a[i]$ and $a[\min]$. 

**Selection sort**

```
\begin{align*}
\text{i} & \quad \text{min} \\
7 \spadesuit & \quad 10 \spadesuit \quad 5 \spadesuit \quad 3 \spadesuit \quad 8 \spadesuit \quad 4 \spadesuit \quad 2 \spadesuit \quad 9 \spadesuit \quad 6 \spadesuit
\end{align*}

remaining entries
```
Selection sort

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

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

![Diagram of selection sort](image)
Selection sort

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

![Selection sort diagram with cards]
Selection sort

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

![Diagram of cards](image_url)
Selection sort

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

![Diagram of selection sort with playing cards]
Selection sort

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

![Diagram of Selection Sort process with playing card images](image-url)
• In iteration $i$, find index $\text{min}$ of smallest remaining entry.
• Swap $a[i]$ and $a[\text{min}]$. 

Selection sort
Selection sort

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

![Card images](image)
Selection sort

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

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

In final order

remaining entries
Selection sort

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

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

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

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

![Diagram showing the process of selection sort with playing cards](image-url)
Selection sort

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

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

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

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

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

![Diagram showing selection sort](image)
Selection sort

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

[Diagram showing cards in a deck with indices and remaining entries labeled.]
Selection sort

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

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

![Diagram of playing cards](image)

*In final order*  
*remaining entries*
Selection sort

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

![Diagram showing selection sort process with playing cards](image)
Selection sort

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

2 ♣️ 3 ♣️ 4 ♣️ 5 ♣️ 6 ♣️ 7 ♣️ 8 ♣️ 10 ♣️
in final order
remaining entries
Selection sort

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

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

![Card deck diagram showing the selection sort process](diagram.png)
Selection sort

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

In final order

remaining entries
Selection sort

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

In final order
Selection sort

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

![Sorted playing cards](image-url)

sorted
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);
        }
    }

    private static boolean less(Comparable v, Comparable w) {
        /* as before */
    }

    private static void exch(Comparable[] a, int i, int j) {
        /* as before */
    }
}
Proposition. Selection sort uses $\sum_{i=0}^{N-2} i + (N-1) \approx N^2/2$ compares and $N$ exchanges.

Running time insensitive to input. Quadratic time, even if input array is sorted. Data movement is minimal. Linear number of exchanges.
Selection sort: animations

20 random items

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

20 partially-sorted items

algorithm position

black in final order

gray not in final order

http://www.sorting-algorithms.com/selection-sort
Elementary Sorting Algorithms

- Sorting review
- Rules of the game
- Selection sort
- Insertion sort
- Shellsort
**Insertion sort**

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

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

\[\text{not yet seen}\]
Selection sort

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

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

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

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

in ascending order  not yet seen
Insertion sort

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

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

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

---

**Diagram:**

Left side: $j$ cards, with 5, 7, 10, and the right side: 3, 8, 4, 2, 9, 6, 9. The card 9 is not yet seen.
Insertion sort

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

- 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

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

![Card Image]

not yet seen
Insertion sort

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

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

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

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

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

- 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

- 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
Insertion sort

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

- 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.
• 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

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

- 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

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

- 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

- 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
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; j--)
                if (less(a[j], a[j-1]))
                    exch(a, j, j-1);
                else break;
    }

    private static boolean less(Comparable v, Comparable w)
    { /* as before */ }

    private static void exch(Comparable[] a, int i, int j)
    { /* as before */ }
}
**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.

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

40 random items

http://www.sorting-algorithms.com/insertion-sort
Insertion sort: best and worst case

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

![Array in ascending order]

**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.

![Array in descending order]
Insertion sort: animation

40 reverse-sorted items

http://www.sorting-algorithms.com/insertion-sort
Def. An **inversion** is a pair of keys that are out of order.

\[
\text{A E E L M O T R X P S}
\]

\[
\text{T-R T-P T-S R-P X-P X-S}
\]

(6 inversions)

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

- Ex 1. A subarray of size 10 appended to a sorted subarray of size \( N \).
- Ex 2. An array of size \( N \) with only 10 entries out of place.

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

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

\[
\text{number of compares} = \text{exchanges} + (N - 1)
\]
Insertion sort: animation

40 partially-sorted items

http://www.sorting-algorithms.com/insertion-sort
ELEMENTARY SORTING ALGORITHMS

- Sorting review
- Rules of the game
- Selection sort
- Insertion sort
- Shellsort
**Idea.** Move entries more than one position at a time by *h*-sorting the array.

An h-sorted array is h interleaved sorted subsequences

Shellsort. [Shell 1959] *h-sort* the array for decreasing seq. of values of *h.*
**h-sorting**

How to \(h\)-sort an array? Insertion sort, with stride length \(h\).

### 3-sorting an array

<table>
<thead>
<tr>
<th>M</th>
<th>O</th>
<th>L</th>
<th>E</th>
<th>E</th>
<th>X</th>
<th>A</th>
<th>S</th>
<th>P</th>
<th>R</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>O</td>
<td>L</td>
<td>M</td>
<td>E</td>
<td>X</td>
<td>A</td>
<td>S</td>
<td>P</td>
<td>R</td>
<td>T</td>
</tr>
<tr>
<td>E</td>
<td>E</td>
<td>L</td>
<td>M</td>
<td>O</td>
<td>X</td>
<td>A</td>
<td>S</td>
<td>P</td>
<td>R</td>
<td>T</td>
</tr>
<tr>
<td>A</td>
<td>E</td>
<td>L</td>
<td>E</td>
<td>O</td>
<td>X</td>
<td>M</td>
<td>S</td>
<td>P</td>
<td>R</td>
<td>T</td>
</tr>
<tr>
<td>A</td>
<td>E</td>
<td>L</td>
<td>E</td>
<td>O</td>
<td>X</td>
<td>M</td>
<td>S</td>
<td>P</td>
<td>R</td>
<td>T</td>
</tr>
<tr>
<td>A</td>
<td>E</td>
<td>L</td>
<td>E</td>
<td>O</td>
<td>P</td>
<td>M</td>
<td>S</td>
<td>X</td>
<td>R</td>
<td>T</td>
</tr>
<tr>
<td>A</td>
<td>E</td>
<td>L</td>
<td>E</td>
<td>O</td>
<td>P</td>
<td>M</td>
<td>S</td>
<td>X</td>
<td>R</td>
<td>T</td>
</tr>
<tr>
<td>A</td>
<td>E</td>
<td>L</td>
<td>E</td>
<td>O</td>
<td>P</td>
<td>M</td>
<td>S</td>
<td>X</td>
<td>R</td>
<td>T</td>
</tr>
<tr>
<td>A</td>
<td>E</td>
<td>L</td>
<td>E</td>
<td>O</td>
<td>P</td>
<td>M</td>
<td>S</td>
<td>X</td>
<td>R</td>
<td>T</td>
</tr>
</tbody>
</table>

### Why insertion sort?

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

**input**

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

**7-sort**

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

**3-sort**

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

**1-sort**

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

**result**

```
A E E L M O P R S T X
```
Proposition. A $g$-sorted array remains $g$-sorted after $h$-sorting it.

7-sort

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

3-sort

\[
\begin{array}{cccccccc}
M & O & L & E & E & X & A & S \\
E & O & L & M & E & X & A & S \\
E & E & L & M & O & X & A & S \\
A & E & L & E & O & X & M & S \\
A & E & L & E & O & X & M & S \\
\end{array}
\]

still 7-sorted
Shell sort: 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.

Interested in learning more?
- See Section 6.8 of Algs, 3rd edition or Volume 3 of Knuth for details.
- Do a JP on the topic.
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, 1093, ...

        while (h >= 1)
        {
            // h-sort the array.
            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;
        }
    }

    private static boolean less(Comparable v, Comparable w)
    { /* as before */ }
    private static boolean void(Comparable[] a, int i, int j)
    { /* as before */ }
}
Shellsort: visual trace

- input
- 40-sorted
- 13-sorted
- 4-sorted
- result
Shellsort: animation

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

50 partially-sorted items

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

algorithm position
h-sorted
current subsequence
other elements
Proposition. The worst-case number of compares used by shellsort with the $3x+1$ increments is $O(N^{3/2})$.

Property. The number of compares used by shellsort with the $3x+1$ increments is at most by a small multiple of $N$ times the # of increments used.

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.
- Tiny, fixed footprint for code (used in 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.

open problem: find a better increment sequence