2.1 Elementary Sorts

- rules of the game
- selection sort
- insertion sort
- shuffling
- comparators
2.1 **Elementary Sorts**

- rules of the game
- selection sort
- insertion sort
- shuffling
- comparators
## Sorting problem

**Ex.** Student records in a university.

<table>
<thead>
<tr>
<th>item</th>
<th>key</th>
<th>name</th>
<th>number 1</th>
<th>number 2</th>
<th>address</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chen</td>
<td>3</td>
<td>A</td>
<td>(991) 878–4944</td>
<td>308 Blair</td>
<td></td>
</tr>
<tr>
<td>Rohde</td>
<td>2</td>
<td>A</td>
<td>(232) 343–5555</td>
<td>343 Forbes</td>
<td></td>
</tr>
<tr>
<td>Gazsi</td>
<td>4</td>
<td>B</td>
<td>(800) 867–5309</td>
<td>101 Brown</td>
<td></td>
</tr>
<tr>
<td>Furia</td>
<td>1</td>
<td>A</td>
<td>(766) 093–9873</td>
<td>101 Brown</td>
<td></td>
</tr>
<tr>
<td>Kanaga</td>
<td>3</td>
<td>B</td>
<td>(898) 122–9643</td>
<td>22 Brown</td>
<td></td>
</tr>
<tr>
<td>Andrews</td>
<td>3</td>
<td>A</td>
<td>(664) 480–0023</td>
<td>097 Little</td>
<td></td>
</tr>
<tr>
<td>Battle</td>
<td>4</td>
<td>C</td>
<td>(874) 088–1212</td>
<td>121 Whitman</td>
<td></td>
</tr>
</tbody>
</table>

**Sort.** Rearrange array of $n$ items in ascending order by key.

<table>
<thead>
<tr>
<th>item</th>
<th>key</th>
<th>name</th>
<th>number 1</th>
<th>number 2</th>
<th>address</th>
</tr>
</thead>
<tbody>
<tr>
<td>Andrews</td>
<td>3</td>
<td>A</td>
<td>(664) 480–0023</td>
<td>097 Little</td>
<td></td>
</tr>
<tr>
<td>Battle</td>
<td>4</td>
<td>C</td>
<td>(874) 088–1212</td>
<td>121 Whitman</td>
<td></td>
</tr>
<tr>
<td>Chen</td>
<td>3</td>
<td>A</td>
<td>(991) 878–4944</td>
<td>308 Blair</td>
<td></td>
</tr>
<tr>
<td>Furia</td>
<td>1</td>
<td>A</td>
<td>(766) 093–9873</td>
<td>101 Brown</td>
<td></td>
</tr>
<tr>
<td>Gazsi</td>
<td>4</td>
<td>B</td>
<td>(800) 867–5309</td>
<td>101 Brown</td>
<td></td>
</tr>
<tr>
<td>Kanaga</td>
<td>3</td>
<td>B</td>
<td>(898) 122–9643</td>
<td>22 Brown</td>
<td></td>
</tr>
<tr>
<td>Rohde</td>
<td>2</td>
<td>A</td>
<td>(232) 343–5555</td>
<td>343 Forbes</td>
<td></td>
</tr>
</tbody>
</table>
Sorting applications

Library of Congress numbers

FedEx packages

playing cards

contacts

Hogwarts houses
Sample sort clients

Goal. Sort any type of data.
Ex 1. Sort strings in alphabetical order.

```java
public class StringSorter {
    public static void main(String[] args) {
        String[] a = StdIn.readStringStrings();
        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 clients

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

**Ex 2.** 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]);
    }
}
```

```bash
% 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 clients

Goal. Sort any type of data.

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

import java.io.File;

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

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

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

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

**Examples.**

<table>
<thead>
<tr>
<th>Video name</th>
<th>Views</th>
</tr>
</thead>
<tbody>
<tr>
<td>'Despacito'[6]</td>
<td>2,993,700,000</td>
</tr>
<tr>
<td>'See You Again'[11]</td>
<td>2,894,000,000</td>
</tr>
<tr>
<td>'Gangnam Style'[17]</td>
<td>803,700,000</td>
</tr>
<tr>
<td>'Baby'[41]</td>
<td>245,400,000</td>
</tr>
<tr>
<td>'Bad Romance'[146]</td>
<td>178,400,000</td>
</tr>
<tr>
<td>'Charlie Bit My Finger'[136]</td>
<td>128,900,000</td>
</tr>
<tr>
<td>'Evolution of Dance'[131]</td>
<td>118,900,000</td>
</tr>
</tbody>
</table>

- **numerical order (descending)**
- **chronological order**
- **lexicographic order**
Total order

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

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

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

Non-examples.

- course prerequisites (violates totality)
- Ro–sham–bo order (violates transitivity)
- predator–prey (violates antisymmetry)
Callbacks

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

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

Callback = reference to executable code.
- Client passes array of objects to sort() function.
- The sort() method calls object’s compareTo() function 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.** A type that defines a set of methods that a class can provide.

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

**Class that implements interface.** Must implement all interface methods.

```java
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
Suppose that the Java architects leave out `implements Comparable<String>` in the class declaration for `String`. What would be the effect?

A. `String.java` won't compile.

B. `StringSorter.java` won’t compile.

C. `Insertion.java` won’t compile.

D. `Insertion.java` will throw an exception.
java.lang.Comparable API

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

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

\[
\begin{align*}
\text{v.compareTo(w)} \leq 0 & \quad \text{means } v \text{ is less than or equal to } w \\
\text{v.compareTo(w)} & \quad \text{less than (return negative integer)} \\
\text{v.compareTo(w)} & \quad \text{equal to (return 0)} \\
\text{v.compareTo(w)} & \quad \text{greater than (return positive integer)}
\end{align*}
\]

**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;
    }
}
```

https://algs4.cs.princeton.edu/12oop/Date.java.html
2.1 Elementary Sorts

- rules of the game
- selection sort
- insertion sort
- shuffling
- comparators
Selection sort demo

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

**Algorithm.** ↑ scans from left to right.

**Invariants.**
- Entries the left of ↑ (including ↑) fixed and in ascending order.
- No entry to right of ↑ is smaller than any entry to the left of ↑.
Selection sort inner loop

To maintain algorithm invariants:

• Move the pointer to the right.

```java
i++;
```

• Identify index of minimum entry on right.

```java
int min = i;
for (int j = i+1; j < n; j++)
    if (less(a[j], a[min]))
        min = j;
```

• Exchange into position.

```java
exch(a, i, min);`
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;
}
```
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)
    { /* see previous slide */ }

    private static void exch(Object[] a, int i, int j)
    { /* see previous slide */ }
}

https://algs4.cs.princeton.edu/21elementary/Selection.java.html
Generic methods

**Oops.** The compiler complains.

```
% javac-algs4 Selection.java
Selection.java:83: warning: [unchecked] unchecked call to compareTo(T) as a member of the raw type java.lang.Comparable
       return (v.compareTo(w) < 0);
          ^
1 warning
```

**Q.** How to silence the compiler?
Generic methods

**Pedantic (type-safe) version.** Compiles without any warnings.

```java
public class SelectionPedantic {
    public static <Key extends Comparable<Key>> void sort(Key[] a)
    { /* as before */ }

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

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


**Remark.** Use type-safe version in system code (but not in lecture).
Selection sort: animations

20 random items

△ algorithm position

- in final order
- not in final order

http://www.sorting-algorithms.com/selection-sort
How many compares does selection sort make to sort an array of $n$ distinct items in reverse order?

A. $\sim n$

B. $\sim \frac{1}{4} n^2$

C. $\sim \frac{1}{2} n^2$

D. $\sim n^2$
Selection sort: mathematical analysis

**Proposition.** Selection sort makes \((n - 1) + (n - 2) + \ldots + 1 + 0 \sim n^2 / 2\) 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[])</td>
<td>(\text{SORTEXAMPLE})</td>
<td>(\text{S})</td>
<td>(\text{O})</td>
<td>(\text{R})</td>
<td>(\text{T})</td>
<td>(\text{E})</td>
<td>(\text{X})</td>
<td>(\text{A})</td>
<td>(\text{M})</td>
<td>(\text{P})</td>
<td>(\text{L})</td>
<td>(\text{E})</td>
</tr>
<tr>
<td>0</td>
<td>6</td>
<td>(\text{SORTEXAMPLE})</td>
<td>(\text{S})</td>
<td>(\text{O})</td>
<td>(\text{R})</td>
<td>(\text{T})</td>
<td>(\text{E})</td>
<td>(\text{X})</td>
<td>(\text{A})</td>
<td>(\text{M})</td>
<td>(\text{P})</td>
<td>(\text{L})</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>(\text{AORTESMAMPLE})</td>
<td>(\text{A})</td>
<td>(\text{O})</td>
<td>(\text{R})</td>
<td>(\text{T})</td>
<td>(\text{E})</td>
<td>(\text{X})</td>
<td>(\text{S})</td>
<td>(\text{M})</td>
<td>(\text{P})</td>
<td>(\text{L})</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
<td>(\text{AERTOSMPLE})</td>
<td>(\text{A})</td>
<td>(\text{E})</td>
<td>(\text{R})</td>
<td>(\text{T})</td>
<td>(\text{O})</td>
<td>(\text{X})</td>
<td>(\text{S})</td>
<td>(\text{M})</td>
<td>(\text{P})</td>
<td>(\text{L})</td>
</tr>
<tr>
<td>3</td>
<td>9</td>
<td>(\text{AETOSMPPLR})</td>
<td>(\text{A})</td>
<td>(\text{E})</td>
<td>(\text{T})</td>
<td>(\text{O})</td>
<td>(\text{X})</td>
<td>(\text{S})</td>
<td>(\text{M})</td>
<td>(\text{P})</td>
<td>(\text{L})</td>
<td>(\text{R})</td>
</tr>
<tr>
<td>4</td>
<td>7</td>
<td>(\text{AELTOPTR})</td>
<td>(\text{A})</td>
<td>(\text{E})</td>
<td>(\text{E})</td>
<td>(\text{L})</td>
<td>(\text{O})</td>
<td>(\text{X})</td>
<td>(\text{S})</td>
<td>(\text{M})</td>
<td>(\text{P})</td>
<td>(\text{T})</td>
</tr>
<tr>
<td>5</td>
<td>7</td>
<td>(\text{AELMXOSPTR})</td>
<td>(\text{A})</td>
<td>(\text{E})</td>
<td>(\text{E})</td>
<td>(\text{L})</td>
<td>(\text{M})</td>
<td>(\text{X})</td>
<td>(\text{S})</td>
<td>(\text{O})</td>
<td>(\text{P})</td>
<td>(\text{T})</td>
</tr>
<tr>
<td>6</td>
<td>8</td>
<td>(\text{AELMOSXPTR})</td>
<td>(\text{A})</td>
<td>(\text{E})</td>
<td>(\text{E})</td>
<td>(\text{L})</td>
<td>(\text{M})</td>
<td>(\text{O})</td>
<td>(\text{S})</td>
<td>(\text{X})</td>
<td>(\text{P})</td>
<td>(\text{T})</td>
</tr>
<tr>
<td>7</td>
<td>10</td>
<td>(\text{AELMOPXSRT})</td>
<td>(\text{A})</td>
<td>(\text{E})</td>
<td>(\text{E})</td>
<td>(\text{L})</td>
<td>(\text{M})</td>
<td>(\text{O})</td>
<td>(\text{P})</td>
<td>(\text{X})</td>
<td>(\text{S})</td>
<td>(\text{T})</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>(\text{AELMOPRSXT})</td>
<td>(\text{A})</td>
<td>(\text{E})</td>
<td>(\text{E})</td>
<td>(\text{L})</td>
<td>(\text{M})</td>
<td>(\text{O})</td>
<td>(\text{P})</td>
<td>(\text{R})</td>
<td>(\text{S})</td>
<td>(\text{T})</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td>(\text{AELMOPRSXT})</td>
<td>(\text{A})</td>
<td>(\text{E})</td>
<td>(\text{E})</td>
<td>(\text{L})</td>
<td>(\text{M})</td>
<td>(\text{O})</td>
<td>(\text{P})</td>
<td>(\text{R})</td>
<td>(\text{S})</td>
<td>(\text{T})</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>(\text{AELMOPRSXT})</td>
<td>(\text{A})</td>
<td>(\text{E})</td>
<td>(\text{E})</td>
<td>(\text{L})</td>
<td>(\text{M})</td>
<td>(\text{O})</td>
<td>(\text{P})</td>
<td>(\text{R})</td>
<td>(\text{S})</td>
<td>(\text{T})</td>
</tr>
</tbody>
</table>

- **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

- rules of the game
- selection sort
- insertion sort
- shuffling
- comparators
Insertion sort demo

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

https://www.youtube.com/watch?v=ROalU379I3U
Insertion sort

Algorithm. ↑ scans from left to right.

Invariants.

- Entries to the left of ↑ (including ↑) are in ascending order.
- Entries to the right of ↑ have not yet been seen.
Insertion sort: inner loop

To maintain algorithm invariants:

- Move the pointer to the right.
  
  ```
  i++;
  ```

- Moving from right to left, exchange `a[i]` with each larger entry to its left.
  
  ```
  for (int j = i; j > 0; j--)
      if (less(a[j], a[j-1]))
          exch(a, j, j-1);
      else break;
  ```
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(Object[] a, int i, int j)
    { /* as before */ }
}
How many compares does insertion sort make to sort an array of $n$ distinct keys in reverse order?

A. $\sim n$
B. $\sim 1/4 \ n^2$
C. $\sim 1/2 \ n^2$
D. $\sim n^2$
**Insertion sort: analysis**

**Worst case.** Insertion sort makes $\sim \frac{1}{2} n^2$ compares and $\sim \frac{1}{2} n^2$ exchanges to sort an array of $n$ distinct keys in reverse order.

**Pf.** Exactly $i$ compares and exchanges in iteration $i$.

$$0 + 1 + 2 + \ldots + (n-1)$$

[Graph showing the complexity of insertion sort withiteration count and comparisons.]
Insertion sort: mathematical analysis

**Average case.** To sort a randomly ordered array with $n$ distinct keys, insertion sort makes $\sim \frac{1}{4} n^2$ compares and $\sim \frac{1}{4} n^2$ exchanges on average.

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

**Average case.** To sort a randomly ordered array with \( n \) distinct keys, insertion sort makes \( \sim \frac{1}{4} n^2 \) compares and \( \sim \frac{1}{4} n^2 \) exchanges on average.

**Pf.** Expect \( \sim \frac{1}{2} i \) compares and \( \sim \frac{1}{2} i \) exchanges in iteration \( i \).

<table>
<thead>
<tr>
<th></th>
<th></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>i</td>
<td>j</td>
<td>a[ ]</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>0</td>
<td>S O R T E X A M P L E</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>1</td>
<td>O S R T E X A M P L E</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>3</td>
<td>O R S T E X A M P L E</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>0</td>
<td>E O R S T E X A M P L E</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>5</td>
<td>E O R S T E X A M P L E</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>0</td>
<td>A E O R S T E X A M P L E</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>2</td>
<td>A E M O R S T E X A M P L E</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>4</td>
<td>A E M O P R S T E X A M P L E</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>2</td>
<td>A E L M O P R S T E X</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>2</td>
<td>A E E L M O P R S T X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

0 + 1 + 2 + … + (n – 1) \( \frac{1}{2} \) entries in black moved one position right for insertion

entries in gray do not move

entry in red is \( a[j] \)

Trace of insertion sort (array contents just after each insertion)
Which is faster in practice to sort an array of \( n \) random items, selection sort or insertion sort?

A. Selection sort.

B. Insertion sort.

C. No significant difference.
Insertion sort: analysis

Best case. Insertion sort makes \( n - 1 \) compares and 0 exchanges to sort an array of \( n \) distinct keys in ascending order.

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

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

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

(6 inversions)

**Def.** A family of arrays is partially sorted if the number of inversions is \( \leq c n \).

- Ex 1. A sorted array.
- Ex 2. A subarray of length 10 appended to a sorted subarray of length \( n \).

**Proposition.** Insertion sort runs in linear time on partially sorted arrays.

**Pf.**

- Number of exchanges in insertion sort = number of inversions.
- Number of compares \( \leq \) number of exchanges + \((n - 1)\).

exchange decreases number of inversions by 1

each compare in iteration \( i \) triggers an exchange (except possibly last one)
Insertion sort: practical improvements

**Half exchanges.** Shift items over (instead of exchanging).
- Eliminates unnecessary data movement.
- No longer uses only `less()` and `exch()` to access data.

\[
\begin{array}{ccccccccccccccc}
\end{array}
\]

**Binary insertion sort.** Use binary search to find insertion point.
- Number of compares \( \sim n \log_2 n \).
- But still a quadratic number of array accesses.

\[
\begin{array}{cccccccccccccccc}
\end{array}
\]

- Binary search for first key > K
2.1 Elementary Sorts

- rules of the game
- selection sort
- insertion sort
- shuffling
- comparators
**Interview Question: Shuffle an Array**

**Goal.** Rearrange array so that result is a uniformly random permutation.

![Card Deck](image-url)
**INTERVIEW QUESTION: SHUFFLE AN ARRAY**

Goal. Rearrange array so that result is a uniformly random permutation.

- all $n!$ permutations equally likely
Shuffle sort

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

<table>
<thead>
<tr>
<th>Card</th>
<th>Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>2♣</td>
<td>0.8003</td>
</tr>
<tr>
<td>3♣</td>
<td>0.9706</td>
</tr>
<tr>
<td>4♣</td>
<td>0.9157</td>
</tr>
<tr>
<td>5♣</td>
<td>0.9649</td>
</tr>
<tr>
<td>6♣</td>
<td>0.1576</td>
</tr>
<tr>
<td>7♣</td>
<td>0.4854</td>
</tr>
<tr>
<td>8♣</td>
<td>0.1419</td>
</tr>
<tr>
<td>9♣</td>
<td>0.4218</td>
</tr>
<tr>
<td>10♣</td>
<td>0.9572</td>
</tr>
</tbody>
</table>
Shuffle sort

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

\[ \begin{array}{cccccccc}
8 & 6 & 9 & 7 & 2 & 4 & 10 & 5 & 3 \\
\text{Clubs} & \text{Clubs} & \text{Clubs} & \text{Clubs} & \text{Clubs} & \text{Clubs} & \text{Clubs} & \text{Clubs} & \text{Clubs} \\
0.1419 & 0.1576 & 0.4218 & 0.4854 & 0.8003 & 0.9157 & 0.9572 & 0.9649 & 0.9706 \\
\end{array} \]
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.
Microsoft antitrust probe by EU. Microsoft agreed to provide a randomized ballot screen for users to select browser.

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.
- **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**: 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.

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;
}
```

browser comparator (fails to implement a total order)
2.1 **Elementary Sorts**

- rules of the game
- selection sort
- insertion sort
- shuffling
- *comparators*
Different orderings

Q. When might we need to define different sort orderings?
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;
    }
}
```

https://algs4.cs.princeton.edu/12oop/Date.java.html
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>
</tr>
</thead>
<tbody>
<tr>
<td>natural order</td>
<td>Now is the time</td>
</tr>
<tr>
<td>case insensitive</td>
<td>is Now the time</td>
</tr>
<tr>
<td>Spanish language</td>
<td>café cafetero cuarto churro nube ñoño</td>
</tr>
<tr>
<td>British phone book</td>
<td>McKinley Mackintosh</td>
</tr>
</tbody>
</table>
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:

- 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;  }
}
```

https://algs4.cs.princeton.edu/21elementary/Insertion.java.html
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;
    ...

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

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

https://algs4.cs.princeton.edu/12oop/Student.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;
  ...

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

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
Insertion.sort(a, Student.byNameOrder());
```

<table>
<thead>
<tr>
<th>Name</th>
<th>ID</th>
<th>Grade</th>
<th>Phone Number</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>

```java
Insertion.sort(a, Student.bySectionOrder());
```

<table>
<thead>
<tr>
<th>Name</th>
<th>ID</th>
<th>Grade</th>
<th>Phone Number</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>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>(800) 867–5309</td>
<td>101 Brown</td>
</tr>
</tbody>
</table>
### Stability

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

```java
Selection.sort(a, Student.byNameOrder());
```

<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>(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>Fox</td>
<td>3</td>
<td>(884) 232–5341</td>
<td>11 Dickinson</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>(800) 867–5309</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>

```java
Selection.sort(a, Student.bySectionOrder());
```

<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>(766) 093–9873</td>
<td>101 Brown</td>
</tr>
<tr>
<td>Rohde</td>
<td>2</td>
<td>(232) 343–5555</td>
<td>343 Forbes</td>
</tr>
<tr>
<td>Chen</td>
<td>3</td>
<td>(991) 878–4944</td>
<td>308 Blair</td>
</tr>
<tr>
<td>Fox</td>
<td>3</td>
<td>(884) 232–5341</td>
<td>11 Dickinson</td>
</tr>
<tr>
<td>Andrews</td>
<td>3</td>
<td>(664) 480–0023</td>
<td>097 Little</td>
</tr>
<tr>
<td>Kanaga</td>
<td>3</td>
<td>(898) 122–9643</td>
<td>22 Brown</td>
</tr>
<tr>
<td>Gazsi</td>
<td>4</td>
<td>(800) 867–5309</td>
<td>101 Brown</td>
</tr>
<tr>
<td>Battle</td>
<td>4</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.
Elementary sorts: quiz 5

Which sorting algorithm(s) are stable?

A. Selection sort.
B. Insertion sort.
C. Both A and B.
D. Neither A nor B.
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>
<tr>
<td></td>
<td></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.
Stability: selection sort

**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);
                }
        }
    }
}
```

<table>
<thead>
<tr>
<th>i</th>
<th>min</th>
<th>0</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td>B1</td>
<td>B2</td>
<td>A</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>A</td>
<td>B2</td>
<td>B1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>A</td>
<td>B2</td>
<td>B1</td>
</tr>
</tbody>
</table>

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