2.1 Elementary Sorts

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

Bogosort

```java
while (!array.isSorted())
    array.permute_randomly();
```

Announcement

Starting Monday, we’ll use iClicker responses as a proxy for attendance
2.1 ELEMENTARY Sorts

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

### Sorting problem

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

<table>
<thead>
<tr>
<th>item</th>
<th>key</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chen</td>
<td>A</td>
</tr>
<tr>
<td>Rohde</td>
<td>A</td>
</tr>
<tr>
<td>Gazi</td>
<td>B</td>
</tr>
<tr>
<td>Furia</td>
<td>A</td>
</tr>
<tr>
<td>Kanaga</td>
<td>B</td>
</tr>
<tr>
<td>Andrews</td>
<td>A</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>item</th>
<th>key</th>
</tr>
</thead>
<tbody>
<tr>
<td>Andrews</td>
<td>A</td>
</tr>
<tr>
<td>Battle</td>
<td>C</td>
</tr>
<tr>
<td>Chen</td>
<td>A</td>
</tr>
<tr>
<td>Furia</td>
<td>A</td>
</tr>
<tr>
<td>Gazi</td>
<td>B</td>
</tr>
<tr>
<td>Kanaga</td>
<td>B</td>
</tr>
<tr>
<td>Rohde</td>
<td>A</td>
</tr>
</tbody>
</table>

### Prerequisites

**Goal.** Sort any type of data (for which sorting is well defined).
- **Ex 1.** Sort random real numbers in ascending order.
- **Ex 2.** Sort strings in alphabetical order.
- **Ex 3.** Sort the files in a given directory by filename.

**Requirement: total order**
- Any two items $v, w$ satisfy $v < w$ or $v = w$ or $v > w$
- There is no cycle of $<$ relationships

---

**Sorting arrays vs. linked lists**

We’ll be exclusively concerned with sorting arrays.

**Q.** Why not study how to sort linked lists?

**A.**
- Most data we’ll want to sort will be in an array anyway.
- If it isn’t, fastest way is to convert to array, sort, convert back.

- Linked lists are typically used for dynamic data.
- Sorting makes sense only for static data.
- But what if we have values coming in dynamically and we want to keep the list sorted at all times?

---

**second half of the course**
Total order: more math-y version

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.

Not transitive. Ro-sham-bo.
Not total. PU course prerequisites.

Modularity and abstraction

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

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

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

```java
private static boolean less(Object v, Object w)
{
  ... // compare v and w
}
```

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;
}
```

Selection sort demo

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

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

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 $a[\min]$.

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 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**

**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.
  ```
  i++; 
  ```

- Identify index of minimum entry on right.
  ```
  \text{int } \text{min} = i; 
  \text{for } (\text{int } j = i+1; j < N; j++) 
  \text{if } (\text{less}(a[j], a[\text{min}])) 
  \text{min} = j; 
  ```

- Exchange into position.
  ```
  \text{exch}(a, i, \text{min}); 
  ```
Selection sort: Java implementation

```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);
        }
    }
}
```

private static boolean less(Comparable v, Comparable w)
{ /* see Comparators section */ }

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

http://algs4.cs.princeton.edu/21elementary/Selection.java.html

Selection sort: animations

20 random items

20 partially-sorted items

Elementary sorts: quiz 1

How many compares does selection sort make to sort an array of \( N \) keys?

A. \( \sim N \)
B. \( \sim \frac{1}{4} N^2 \)
C. \( \sim \frac{1}{2} N^2 \)
D. \( \sim N^2 \)
E. I don't know.
Selection sort: mathematical analysis

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

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

**Insertion sort demo**

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

**Algorithm.**

- \(\uparrow\) scans from left to right.

**Invariants.**

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

To maintain algorithm invariants:

- Move the pointer to the right.

```
    i++; // move the pointer to the right.
```

- 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;
```

**Insertion sort: Java implementation**

```java
public class InsertionSort {
    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) {
        return v.compareTo(w) < 0; // as before }
    }

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

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

**Insertion sort: mathematical analysis**

**Proposition.** To sort a randomly-ordered array with distinct keys, insertion sort uses \( \frac{3}{2} N^2 \) compares and \( \frac{1}{2} N^2 \) exchanges on average.

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

```
  i   j  0  1  2  3  4  5  6  7  8  9 10
  0  0  O  S  E  R  T  E  X  A  M  P  L  E
  1  1  0  S  E  R  T  E  X  A  M  P  L  E
  2  1  0  S  E  R  T  E  X  A  M  P  L  E
  3  3  0  E  R  S  T  E  X  A  M  P  L  E
  4  0  E  R  S  T  E  X  A  M  P  L  E
  5  5  E  R  S  T  E  X  A  M  P  L  E
  6  0  A  E  R  S  T  E  X  A  M  P  L  E
  7  2  A  E  R  S  T  E  X  A  M  P  L  E
  8  4  A  E  R  S  T  E  X  A  M  P  L  E
  9  2  A  E  R  S  T  E  X  A  M  P  L  E
 10  2  A  E  R  S  T  E  X  A  M  P  L  E
```

- entries in gray do not move
- entries in red are not moved
- entries in black moved one position right for insertion
- algorithm position in order
- not yet seen
Elementary sorts: quiz 2

How many compares does insertion sort make to sort an array of $N$ distinct keys in reverse order?

A. $\sim N$
B. $\sim \frac{1}{4} N^2$
C. $\sim \frac{1}{2} N^2$
D. $\sim N^2$
E. I don’t know.

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: partially-sorted arrays**

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

AEELMOTXPS


(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.

\[
\text{number of compares } \geq \text{ exchanges } + (N - 1)
\]

**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.

ACHI MNPQXY KINARY

**Binary insertion sort.** Use binary search to find insertion point.

- Number of compares \( \sim N \log N \).
- But still a quadratic number of array accesses.

ACHI[MN]NPQXY KINARY

**Elementary sorts: quiz 3**

Which is faster in practice, selection sort or insertion sort?

- **A.** Selection sort.
- **B.** Insertion sort.
- **C.** No significant difference.
- **D.** I don’t know.

Also faster in theory if our cost model incorporates the assumption that comparing two objects is almost always slower than swapping two pointers.
Callbacks

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

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

A. Client object must implement an interface (Comparable).
   - Client passes array of objects to sort() function.
   - The sort() function calls object's compareTo() method as needed.

*This is a callback.* Client calls sort() and sort() calls client code back.

Comparable interface: overview

```java
public class StringSorter
{
    public static void main(String[] args)
    {
        String[] a = StdIn.readAllStrings();
        Insertion.sort(a);
        StdOut.println(a);
    }
}
```

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

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

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

Elementary sorts: quiz 1

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 a run-time exception.
E. I don't know.
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).

Built-in comparable types. Integer, Double, String, Date, File, ...
User-defined comparable types. Implement the Comparable interface.

Review: Selection sort: Java implementation

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 */
   }
}

http://algs4.cs.princeton.edu/13arrays/Selection.java.html

Generic methods

Oops. The compiler complains.

% javac Selection.java
Note: Selection.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.

% javac -Xlint:unchecked Selection.java
Selection.java:8: 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).

Sort music library by artist

Comparable interface: review

Comparable interface: sort using a type's natural order.

```java
public class Date implements Comparable<Date> {
    private final int month, day, year;
    public Date(int m, int d, int y) {
        month = m;
        day = d;
        year = y;
    }
    public int compareTo(Date that) {
        if (this.year < that.year) return -1;
        if (this.year > that.year) return +1;
        if (this.month < that.month) return -1;
        if (this.month > that.month) return +1;
        if (this.day < that.day) return -1;
        if (this.day > that.day) return +1;
        return 0;
    }
}
```
Comparator interface

Comparator interface: sort using an alternate order.

```java
public interface Comparator<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</td>
</tr>
<tr>
<td>British phone book</td>
<td>McKinley Mackintosh</td>
</tr>
</tbody>
</table>

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;
    }
}
```

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: 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;
    ... one Comparator for the class
    public static Comparator<Student> nameOrder() {
        return new NameOrder();
    }
    public 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 implements Comparator<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.

Stability

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

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

<table>
<thead>
<tr>
<th>Student</th>
<th>Phone Number</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>Andrews</td>
<td>684-480-0023</td>
<td>097 Little</td>
</tr>
<tr>
<td>Battle</td>
<td>874-088-1212</td>
<td>121 Whitman</td>
</tr>
<tr>
<td>Chen</td>
<td>981-878-4944</td>
<td>308 Blair</td>
</tr>
<tr>
<td>Fox</td>
<td>884-232-5341</td>
<td>11 Dickinson</td>
</tr>
<tr>
<td>Furia</td>
<td>760-993-9873</td>
<td>101 Brown</td>
</tr>
<tr>
<td>Gazi</td>
<td>981-878-4944</td>
<td>308 Blair</td>
</tr>
<tr>
<td>Kanaga</td>
<td>880-867-5309</td>
<td>22 Brown</td>
</tr>
<tr>
<td>Rohda</td>
<td>232-343-5555</td>
<td>343 Forbes</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Student</th>
<th>Phone Number</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>Andrews</td>
<td>684-480-0023</td>
<td>097 Little</td>
</tr>
<tr>
<td>Battle</td>
<td>874-088-1212</td>
<td>121 Whitman</td>
</tr>
<tr>
<td>Chen</td>
<td>981-878-4944</td>
<td>308 Blair</td>
</tr>
<tr>
<td>Fox</td>
<td>884-232-5341</td>
<td>11 Dickinson</td>
</tr>
<tr>
<td>Furia</td>
<td>760-993-9873</td>
<td>101 Brown</td>
</tr>
<tr>
<td>Gazi</td>
<td>981-878-4944</td>
<td>308 Blair</td>
</tr>
<tr>
<td>Kanaga</td>
<td>880-867-5309</td>
<td>22 Brown</td>
</tr>
<tr>
<td>Rohda</td>
<td>232-343-5555</td>
<td>343 Forbes</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 4

Which sorting algorithms are stable?

A. Selection sort.
B. Insertion sort.
C. Both A and B.
D. Neither A nor B.
E. I don’t know.

Stability

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

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

<table>
<thead>
<tr>
<th>Student</th>
<th>Phone Number</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>Andrews</td>
<td>684-480-0023</td>
<td>097 Little</td>
</tr>
<tr>
<td>Battle</td>
<td>874-088-1212</td>
<td>121 Whitman</td>
</tr>
<tr>
<td>Chen</td>
<td>981-878-4944</td>
<td>308 Blair</td>
</tr>
<tr>
<td>Fox</td>
<td>884-232-5341</td>
<td>11 Dickinson</td>
</tr>
<tr>
<td>Furia</td>
<td>760-993-9873</td>
<td>101 Brown</td>
</tr>
<tr>
<td>Gazi</td>
<td>981-878-4944</td>
<td>308 Blair</td>
</tr>
<tr>
<td>Kanaga</td>
<td>880-867-5309</td>
<td>22 Brown</td>
</tr>
<tr>
<td>Rohda</td>
<td>232-343-5555</td>
<td>343 Forbes</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Student</th>
<th>Phone Number</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>Andrews</td>
<td>684-480-0023</td>
<td>097 Little</td>
</tr>
<tr>
<td>Battle</td>
<td>874-088-1212</td>
<td>121 Whitman</td>
</tr>
<tr>
<td>Chen</td>
<td>981-878-4944</td>
<td>308 Blair</td>
</tr>
<tr>
<td>Fox</td>
<td>884-232-5341</td>
<td>11 Dickinson</td>
</tr>
<tr>
<td>Furia</td>
<td>760-993-9873</td>
<td>101 Brown</td>
</tr>
<tr>
<td>Gazi</td>
<td>981-878-4944</td>
<td>308 Blair</td>
</tr>
<tr>
<td>Kanaga</td>
<td>880-867-5309</td>
<td>22 Brown</td>
</tr>
<tr>
<td>Rohda</td>
<td>232-343-5555</td>
<td>343 Forbes</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 sort(Comparable[] a) {
        int N = a.length;
        for (int i = 1; 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 1 2 3 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>A A A A B</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>A A A A B</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>A A A A B</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>A A A A B</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>A A A A B</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 sort(Comparable[] a) {
        int N = a.length;
        for (int i = 0; i < N; i++)
            int min = i;
            for (int j = i+1; j < N; j++)
                if (less(a[j], a[min]))
                    min = j;
            exch(a, i, min);
    }
}
```

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

Interview question: shuffle an array

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

```
[1 2 3 4 5 6 7 8 9 0]
```

The goal is to `shuffle` the elements of the array so that all permutations are equally likely.
Interview question: shuffle an array

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

Shuffling by sorting

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

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
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 shuffling-by-sorting 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

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


War story (online poker)

Texas hold’em poker. Software must shuffle electronic cards.

**War story (online poker)**

**Shuffling algorithm in FAQ at www.planetpoker.com**

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

**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!

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

— Robert R. Coveyou

---

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

Bug 2. Shuffle not uniform (should be between 1 and 52).

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

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