Acknowledgement: The course slides are adapted from the slides prepared by R. Sedgewick and K. Wayne of Princeton University.
Sorting review
Rules of the game
Selection sort
Insertion sort
Shellsort
Mergesort
Quicksort
Heapsort
Sorting Algorithms

- Sorting review
- Rules of the game
  - Selection sort
  - Insertion sort
  - Shellsort
  - Mergesort
  - Quicksort
- Heapsort
**Sorting problem**

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

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

Goal. Sort any type of data.

Ex 1. Sort random real numbers in ascending order.

```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

seems artificial, but stay tuned for an application
Sample sort client

**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
**Sample sort client**

**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());
   }
}
```

% java FileSorter .
Insertion.class
Insertion.java
InsertionX.class
InsertionX.java
Selection.class
Selection.java
Shell.class
Shell.java
ShellX.class
ShellX.java
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);
}

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;
}

key point: no dependence on File data type
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.
- ...
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.
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;
   }
}
```

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;
}
```
Sorting Algorithms

- Sorting review
- Rules of the game
- Selection sort
- Insertion sort
- Shellsort
- Mergesort
- Quicksort
- Heapsort
Selection sort

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

[Diagram of playing cards showing remaining entries and highlighting one card]
• 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 $a[\text{min}]$. 

![Selection Sort Diagram](image-url)
Selection sort

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

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

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

In final order

remaining entries
Selection sort

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

[Picture of playing cards showing the process of selection sort]

in final order

remaining entries
Selection sort

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

![Image of playing cards illustrating selection sort](image-url)
Selection sort

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

\[ \text{in final order} \quad \text{remaining entries} \]
Selection sort

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

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

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

![Image of playing cards showing selection sort process]
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 $a[\text{min}]$. 

![Selection sort diagram]

- 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 diagram](image-url)
Selection sort

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

![Image showing selection sort process]
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 $a[\text{min}]$.

![Diagram showing the process of selection sort with playing cards representing the indices and values.](image-url)

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

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

![Diagram of playing cards showing in final order and remaining entries]
Selection sort

- 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]
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 $a[\min]$. 

![Image of playing cards](image-url)

- 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 diagram](image)
Selection sort

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

Selection sort
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 $a[\min]$. 

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

in final order

remaining entries
Selection sort

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

[Card images in order from 2 to 10 of clubs]

in final order
Selection sort

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

![Sorted cards](image)
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) {
        /* as before */
    }

    private static void exch(Comparable[] a, int i, int j) {
        /* as before */
    }
}
```
**Selection sort: mathematical analysis**

**Proposition.** Selection sort uses \((N - 1) + (N - 2) + \ldots + 1 + 0 \sim N^2/2\) compares and \(N\) exchanges.

<table>
<thead>
<tr>
<th>(i)</th>
<th>(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></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>(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></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>6</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>1</td>
<td>4</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>2</td>
<td>10</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>3</td>
<td>9</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>4</td>
<td>7</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>5</td>
<td>7</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>6</td>
<td>8</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>7</td>
<td>10</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>8</td>
<td>8</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>9</td>
<td>9</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>10</td>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

**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
Sorting Algorithms

- Sorting review
  - Rules of the game
  - Selection sort
- Insertion sort
  - Shellsort
  - Mergesort
  - Quicksort
- Heapsort
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.
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**

![Diagram](image-url)
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**

[Diagram showing the insertion sort process with cards in ascending order and those not yet seen]
• 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
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 diagram with cards]

not yet seen
• 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.
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.
• In iteration $i$, swap $a[i]$ with each larger entry to its left.

**Insertion sort**

- Cards: 2, 3, 4, 5, 7, 8, 10, 9, 6, 9
- In ascending order: 2, 3, 4, 5, 7, 8, 10
- Not yet seen: 9, 6, 9

![Playing cards illustrating the insertion sort algorithm](image)
• 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 ascending order

not yet seen
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.
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 */  }
}
**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 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>1</td>
<td>0</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>2</td>
<td>1</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>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 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 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 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 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 X 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 X 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>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>

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](Image)

**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](Image)
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.

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
Sorting Algorithms

- Sorting review
  - Rules of the game
  - Selection sort
  - Insertion sort
  - Shellsort
  - Mergesort
  - Quicksort
- Heapsort
**Shellsort overview**

**Idea.** Move entries more than one position at a time by *h*-sorting the array.

an h-sorted array is h interleaved sorted subsequences

<table>
<thead>
<tr>
<th>h = 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>L E E A M H L E P S O L T S X R</td>
</tr>
<tr>
<td>L   M   P   T</td>
</tr>
<tr>
<td>E   H   S   S</td>
</tr>
<tr>
<td>E   L   O   X</td>
</tr>
<tr>
<td>A   E   L   R</td>
</tr>
</tbody>
</table>

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

**input**

<table>
<thead>
<tr>
<th>SHELLSORTEXAMPLE</th>
</tr>
</thead>
<tbody>
<tr>
<td>13-sort</td>
</tr>
<tr>
<td>PHELLSORTEXAMPLE</td>
</tr>
<tr>
<td>4-sort</td>
</tr>
<tr>
<td>LEEAMHLEPSOLTSXR</td>
</tr>
<tr>
<td>1-sort</td>
</tr>
<tr>
<td>AE EEHLLLMMOPRSSSTX</td>
</tr>
</tbody>
</table>
How to \( h \)-sort an array? Insertion sort, with stride length \( h \).

**Why insertion sort?**

- Big increments \( \Rightarrow \) small subarray.
- Small increments \( \Rightarrow \) 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
A E E L O X M S P R T
A E E L O X M S P R T
A E E L O P M S X R T
A E E L O P M S X R T
A E E L O P M S X R T
```

**1-sort**

```
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
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
```
Shellsort: intuition

Proposition. A $g$-sorted array remains $g$-sorted after $h$-sorting it.
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.

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
Shell sort: animation

50 random items

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

50 partially-sorted items

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

**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.

<table>
<thead>
<tr>
<th>N</th>
<th>compares</th>
<th>$N^{1.289}$</th>
<th>$2.5 N \lg N$</th>
</tr>
</thead>
<tbody>
<tr>
<td>5,000</td>
<td>93</td>
<td>58</td>
<td>106</td>
</tr>
<tr>
<td>10,000</td>
<td>209</td>
<td>143</td>
<td>230</td>
</tr>
<tr>
<td>20,000</td>
<td>467</td>
<td>349</td>
<td>495</td>
</tr>
<tr>
<td>40,000</td>
<td>1022</td>
<td>855</td>
<td>1059</td>
</tr>
<tr>
<td>80,000</td>
<td>2266</td>
<td>2089</td>
<td>2257</td>
</tr>
</tbody>
</table>

measured in thousands

**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.
Sorting Algorithms

- Sorting review
  - Rules of the game
  - Selection sort
  - Insertion sort
  - Shellsort
  - Mergesort
  - Quicksort
- Heapsort
Mergesort

Basic plan.

- Divide array into two halves.
- **Recursively** sort each half.
- Merge two halves.

Mergesort overview
**Abstract in-place merge**

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

<table>
<thead>
<tr>
<th></th>
<th>lo</th>
<th>mid</th>
<th>mid+1</th>
<th>hi</th>
</tr>
</thead>
<tbody>
<tr>
<td>a[]</td>
<td>E</td>
<td>E</td>
<td>G</td>
<td>M</td>
</tr>
</tbody>
</table>

**copy to auxiliary array**

<table>
<thead>
<tr>
<th>aux[]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>
Abstract in-place merge

**Goal.** Given two sorted subarrays $a[lo]$ to $a[mid]$ and $a[mid+1]$ to $a[hi]$, replace with sorted subarray $a[lo]$ to $a[hi]$. 

---

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

---

<table>
<thead>
<tr>
<th>$aux[]$</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
</tr>
<tr>
<td>E</td>
</tr>
<tr>
<td>G</td>
</tr>
<tr>
<td>M</td>
</tr>
<tr>
<td>R</td>
</tr>
<tr>
<td>A</td>
</tr>
<tr>
<td>C</td>
</tr>
<tr>
<td>E</td>
</tr>
<tr>
<td>R</td>
</tr>
<tr>
<td>T</td>
</tr>
</tbody>
</table>
**Abstract in-place merge**

**Goal.** Given two sorted subarrays $a[lo]$ to $a[mid]$ and $a[mid+1]$ to $a[hi]$, replace with sorted subarray $a[lo]$ to $a[hi]$.

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

**compare minimum in each subarray**

<table>
<thead>
<tr>
<th>aux[]</th>
<th>E</th>
<th>E</th>
<th>G</th>
<th>M</th>
<th>R</th>
<th>A</th>
<th>C</th>
<th>E</th>
<th>R</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>j</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**Goal.** Given two sorted subarrays \( a[lo] \) to \( a[mid] \) and \( a[mid+1] \) to \( a[hi] \), replace with sorted subarray \( a[lo] \) to \( a[hi] \).

**Abstract in-place merge**

Compare minimum in each subarray

\[
\begin{array}{cccccccccccc}
  a[] & A & E & G & M & R & A & C & E & R & T \\
  \[k\] &   &   &   &   &   &   &   &   &   &   \\
\end{array}
\]

\[
\begin{array}{cccccccccccc}
  \text{aux[]} & E & E & G & M & R & A & C & E & R & T \\
  \[i\] &   &   &   &   &   &   &   &   &   &   \\
  \[j\] &   &   &   &   &   &   &   &   &   &   \\
\end{array}
\]
**Abstract in-place merge**

**Goal.** Given two sorted subarrays $a[lo]$ to $a[mid]$ and $a[mid+1]$ to $a[hi]$, replace with sorted subarray $a[lo]$ to $a[hi]$.

- $a[]$:
  
  | A | E | G | M | R | A | C | E | R | T |
  |
  | k |

- aux[]:
  
  | E | E | G | M | R | A | C | E | R | T |
  |
  | i | j |
**Abstract in-place merge**

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

```
\[\begin{array}{cccccccccccc}
a & A & C & G & M & R & A & C & E & R & T \\
\end{array}\]
```

**compare minimum in each subarray**

```
\[\begin{array}{cccccccccccc}
aux & E & E & G & M & R & A & C & E & R & T \\
\end{array}\]
```
**Abstract in-place merge**

**Goal.** Given two sorted subarrays a[lo] to a[mid] and a[mid+1] to a[hi], replace with sorted subarray a[lo] to a[hi].
Abstract in-place merge

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

```
<table>
<thead>
<tr>
<th>A</th>
<th>C</th>
<th>E</th>
<th>M</th>
<th>R</th>
<th>A</th>
<th>C</th>
<th>E</th>
<th>R</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>k</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

**compare minimum in each subarray**

```
<table>
<thead>
<tr>
<th>E</th>
<th>E</th>
<th>G</th>
<th>M</th>
<th>R</th>
<th>A</th>
<th>C</th>
<th>E</th>
<th>R</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>E</th>
<th>E</th>
<th>G</th>
<th>M</th>
<th>R</th>
<th>A</th>
<th>C</th>
<th>E</th>
<th>R</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>j</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
**Abstract in-place merge**

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

- \( a[] \)
  - A C E M R A C E R T
  - k

- \( \text{aux[]} \)
  - E E G M R A C E R T
  - i
  - j

*compare minimum in each subarray*
**Abstract in-place merge**

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

---

**Diagram: Abstract in-place merge**

- **a[]:**
  
  \[\text{A C E E R A C E R T}\]

  
  \(k\)

- **compare minimum in each subarray**

  **aux[]:**
  
  \[\text{E E G M R A C E R T}\]

  
  \(i\) \hspace{1cm} \(j\)
**Abstract in-place merge**

**Goal.** Given two sorted subarrays \(a[lo] \) to \(a[mid]\) and \(a[mid+1]\) to \(a[hi]\), replace with sorted subarray \(a[lo] \) to \(a[hi]\).
Abstract in-place merge

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

\[
\begin{array}{cccccccccc}
\text{a[]} & A & C & E & E & \text{E} & \text{E} & A & C & E & R & T \\
\text{aux[]} & E & E & G & M & R & A & C & E & R & T \\
\end{array}
\]

**compare minimum in each subarray**

\[
\begin{array}{cccccccccc}
& i & & & & & & j & & & & \\
\end{array}
\]
**Abstract in-place merge**

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

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>C</th>
<th>E</th>
<th>E</th>
<th>E</th>
<th>A</th>
<th>C</th>
<th>E</th>
<th>R</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<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>(k)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**compare minimum in each subarray**

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


**Abstract in-place merge**

**Goal.** Given two sorted subarrays `a[lo]` to `a[mid]` and `a[mid+1]` to `a[hi]`, replace with sorted subarray `a[lo]` to `a[hi]`.

```
<table>
<thead>
<tr>
<th>a[]</th>
<th>A</th>
<th>C</th>
<th>E</th>
<th>E</th>
<th>E</th>
<th>G</th>
<th>C</th>
<th>E</th>
<th>R</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>k</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

**compare minimum in each subarray**

```
<table>
<thead>
<tr>
<th>aux[]</th>
<th>E</th>
<th>E</th>
<th>G</th>
<th>M</th>
<th>R</th>
<th>A</th>
<th>C</th>
<th>E</th>
<th>R</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>j</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
**Abstract in-place merge**

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

![Diagram showing comparison of minimum elements in subarrays and updating the arrays](image)

**compare minimum in each subarray**

- In the `a[]` array, the minimum is located at index `k`.
- In the `aux[]` array, the minimum is located at index `j`.

The merging process involves comparing the minimum elements from each subarray and updating the array accordingly.
**Abstract in-place merge**

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

![Diagram showing comparison and merging process]

**compare minimum in each subarray**
Abstract in-place merge

**Goal.** Given two sorted subarrays `a[lo]` to `a[mid]` and `a[mid+1]` to `a[hi]`, replace with sorted subarray `a[lo]` to `a[hi].`

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

**compare minimum in each subarray**

<table>
<thead>
<tr>
<th>aux[]</th>
<th>E</th>
<th>E</th>
<th>G</th>
<th>M</th>
<th>R</th>
<th>A</th>
<th>C</th>
<th>E</th>
<th>R</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>i</td>
<td>i</td>
<td>i</td>
<td>i</td>
<td>i</td>
<td>j</td>
</tr>
</tbody>
</table>

**Abstract in-place merge**

**Goal.** Given two sorted subarrays $a[lo]$ to $a[mid]$ and $a[mid+1]$ to $a[hi]$, replace with sorted subarray $a[lo]$ to $a[hi]$.

**Diagram:**

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

compare minimum in each subarray

 aux[]   E   E   G   M   R   A   C   E   R   T
       i   j
```
Abstract in-place merge

**Goal.** Given two sorted subarrays $a[lo]$ to $a[mid]$ and $a[mid+1]$ to $a[hi]$, replace with sorted subarray $a[lo]$ to $a[hi]$.

One subarray exhausted, take from other

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

<table>
<thead>
<tr>
<th>aux[]</th>
<th>E</th>
<th>E</th>
<th>G</th>
<th>M</th>
<th>R</th>
<th>A</th>
<th>C</th>
<th>E</th>
<th>R</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>i</td>
<td></td>
<td>j</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Abstract in-place merge

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

One subarray exhausted, take from other

\[
\begin{array}{cccccccc}
\text{a[]} & A & C & E & E & E & G & M & R \\
\text{aux[]} & E & E & G & M & R & A & C & E & R & T \\
\end{array}
\]
**Abstract in-place merge**

**Goal.** Given two sorted subarrays $a[lo]$ to $a[mid]$ and $a[mid+1]$ to $a[hi]$, replace with sorted subarray $a[lo]$ to $a[hi]$.

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

**one subarray exhausted, take from other**

| $aux[]$ | E | E | G | M | R | A | C | E | R | T |
|         |   |   |   |   |   |   |   |   |   |   |
|         |   |   |   |   |   |   |   |   |   |   |
|         |   |   |   |   |   |   |   |   |   |   |
|         |   |   |   |   |   |   |   |   |   |   |
|         |   |   |   |   |   |   |   |   |   |   |
|         |   |   |   |   |   |   |   |   |   |   |

$k$  
i  
j
**Abstract in-place merge**

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

---

**One subarray exhausted, take from other**

```plaintext
<table>
<thead>
<tr>
<th>a[]</th>
<th>A</th>
<th>C</th>
<th>E</th>
<th>E</th>
<th>E</th>
<th>G</th>
<th>M</th>
<th>R</th>
<th>R</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>k</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>aux[]</th>
<th>E</th>
<th>E</th>
<th>G</th>
<th>M</th>
<th>R</th>
<th>A</th>
<th>C</th>
<th>E</th>
<th>R</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>i</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>j</td>
</tr>
</tbody>
</table>
```
**Abstract in-place merge**

**Goal.** Given two sorted subarrays `a[lo]` to `a[mid]` and `a[mid+1]` to `a[hi]`, replace with sorted subarray `a[lo]` to `a[hi].`
**Abstract in-place merge**

**Goal.** Given two sorted subarrays `a[lo]` to `a[mid]` and `a[mid+1]` to `a[hi]`, replace with sorted subarray `a[lo]` to `a[hi].`
**Merging**

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

**A.** Use an auxiliary array.

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

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

<table>
<thead>
<tr>
<th>0</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>A</td>
</tr>
<tr>
<td>1</td>
<td>A</td>
</tr>
<tr>
<td>2</td>
<td>A</td>
</tr>
<tr>
<td>3</td>
<td>A</td>
</tr>
<tr>
<td>4</td>
<td>A</td>
</tr>
<tr>
<td>5</td>
<td>A</td>
</tr>
<tr>
<td>6</td>
<td>A</td>
</tr>
<tr>
<td>7</td>
<td>A</td>
</tr>
<tr>
<td>8</td>
<td>A</td>
</tr>
<tr>
<td>9</td>
<td>A</td>
</tr>
</tbody>
</table>

**merged result**

A C E E E G M R R T

Abstract in-place merge trace
private static void merge(Comparable[] a, Comparable[] aux, int lo, int mid, int hi) {
    assert isSorted(a, lo, mid);  // precondition: a[lo..mid] sorted
    assert isSorted(a, mid+1, hi); // precondition: a[mid+1..hi] sorted

    for (int k = lo; k <= hi; k++)
        aux[k] = a[k];

    int i = lo, j = mid+1;
    for (int k = lo; k <= hi; k++)
        {  // merge
            if      (i > mid)      a[k] = aux[j++];
            else if (j > hi)      a[k] = aux[i++];
            else if (less(aux[j], aux[i])) a[k] = aux[j++];
            else                  a[k] = aux[i++];
        }

    assert isSorted(a, lo, hi);  // postcondition: a[lo..hi] sorted
}

int i = lo, j = mid+1;
for (int k = lo; k <= hi; k++)
    {  // merge
        if      (i > mid)      a[k] = aux[j++];
        else if (j > hi)      a[k] = aux[i++];
        else if (less(aux[j], aux[i])) a[k] = aux[j++];
        else                  a[k] = aux[i++];
    }
Mergesort: Java implementation

```java
public class Merge {
    private static void merge(Comparable[] a, Comparable[] aux, int lo, int mid, int hi) {
        /* as before */
    }

    private static void sort(Comparable[] a, Comparable[] aux, int lo, int hi) {
        if (hi <= lo) return;
        int mid = lo + (hi - lo) / 2;
        sort(a, aux, lo, mid);
        sort(a, aux, mid+1, hi);
        merge(a, aux, lo, mid, hi);
    }

    public static void sort(Comparable[] a) {
        aux = new Comparable[a.length];
        sort(a, aux, 0, a.length - 1);
    }
}
```
Mergesort: trace

Trace of merge results for top-down mergesort

<table>
<thead>
<tr>
<th>Merge Call</th>
<th>lo</th>
<th>hi</th>
</tr>
</thead>
<tbody>
<tr>
<td>merge(a, 0, 0, 1)</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>merge(a, 2, 2, 3)</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>merge(a, 0, 1, 3)</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>merge(a, 4, 4, 5)</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>merge(a, 6, 6, 7)</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>merge(a, 4, 5, 7)</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>merge(a, 0, 3, 7)</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>merge(a, 8, 8, 9)</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>merge(a, 10, 10, 11)</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>merge(a, 8, 9, 11)</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>merge(a, 12, 12, 13)</td>
<td>12</td>
<td>12</td>
</tr>
<tr>
<td>merge(a, 14, 14, 15)</td>
<td>14</td>
<td>14</td>
</tr>
<tr>
<td>merge(a, 12, 13, 15)</td>
<td>12</td>
<td>13</td>
</tr>
<tr>
<td>merge(a, 8, 11, 15)</td>
<td>8</td>
<td>11</td>
</tr>
<tr>
<td>merge(a, 0, 7, 15)</td>
<td>0</td>
<td>7</td>
</tr>
</tbody>
</table>

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

result after recursive call
Mergesort: animation

50 random items

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

50 reverse-sorted items

http://www.sorting-algorithms.com/merge-sort
Mergesort: empirical analysis

Running time estimates:
- Laptop executes $10^8$ compares/second.
- Supercomputer executes $10^{12}$ compares/second.

<table>
<thead>
<tr>
<th>computer</th>
<th>insertion sort ($N^2$)</th>
<th>mergesort ($N \log N$)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>thousand</td>
<td>million</td>
</tr>
<tr>
<td>home</td>
<td>instant</td>
<td>2.8 hours</td>
</tr>
<tr>
<td>super</td>
<td>instant</td>
<td>1 second</td>
</tr>
</tbody>
</table>

Bottom line. Good algorithms are better than supercomputers.
Proposition. Mergesort uses at most $N \lg N$ compares and $6N \lg N$ array accesses to sort any array of size $N$.

Pf sketch. The number of compares $C(N)$ and array accesses $A(N)$ to mergesort an array of size $N$ satisfy the recurrences:

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

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

We solve the recurrence when $N$ is a power of 2.

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

Pf 1. [assuming \( N \) is a power of 2]

\[
\begin{align*}
D(N) &= N \\
D(N/2) &= N \\
D(N/2) &= N \\
D(N/2) &= N \\
D(N/4) &= N \\
D(N/4) &= N \\
D(N/4) &= N \\
D(N/4) &= N \\
D(N/2^k) &= N \\
D(2) &= N \\
D(2) &= N \\
D(2) &= N \\
D(2) &= N \\
D(2) &= N \\
D(2) &= N \\
D(2) &= N \\
N/2 &= N
\end{align*}
\]

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

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

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

\[
D(N) = 2 \, D(N/2) + N \\
D(N) / N = 2 \, D(N/2) / N + 1 \\
= D(N/2) / (N/2) + 1 \\
= D(N/4) / (N/4) + 1 + 1 \\
= D(N/8) / (N/8) + 1 + 1 + 1 \\
\ldots \\
= D(N/N) / (N/N) + 1 + 1 + \ldots + 1 \\
= \lg N
\]

given

divide both sides by \( N \)
algebra
apply to first term
apply to first term again

stop applying, \( D(1) = 0 \)
Proposition. If $D(N)$ satisfies $D(N) = 2D(N/2) + N$ for $N > 1$, with $D(1) = 0$, then $D(N) = N \log N$.

Pf 3. [assuming $N$ is a power of 2]

- Base case: $N = 1$.
- Inductive hypothesis: $D(N) = N \log N$.
- Goal: show that $D(2N) = (2N) \log (2N)$.

\[
D(2N) = 2D(N) + 2N \\
= 2N \log N + 2N \\
= 2N(\log (2N) - 1) + 2N \\
= 2N \log (2N)
\]
Mergesort analysis: memory

Proposition. Mergesort uses extra space proportional to $N$.

Pf. The array $\text{aux}[]$ needs to be of size $N$ for the last merge.

Def. A sorting algorithm is in-place if it uses $\leq c \log N$ extra memory.

Ex. Insertion sort, selection sort, shellsort.

Challenge for the bored. In-place merge. [Kronrod, 1969]
Mergesort: practical improvements

Use insertion sort for small subarrays.

- Mergesort has too much overhead for tiny subarrays.
- Cutoff to insertion sort for ≈ 7 items.

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

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

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

Eliminate the copy to the auxiliary array. Save time (but not space) by switching the role of the input and auxiliary array in each recursive call.

```java
private static void merge(Comparable[] a, Comparable[] aux, int lo, int mid, int hi) {
    int i = lo, j = mid+1;
    for (int k = lo; k <= hi; k++) {
        if      (i > mid)           aux[k] = a[j++];
        else if (j > hi)            aux[k] = a[i++];
        else if (less(a[j], a[i])) aux[k] = a[j++];
        else                        aux[k] = a[i++];
    }
}

private static void sort(Comparable[] a, Comparable[] aux, int lo, int hi) {
    if (hi <= lo) return;
    int mid = lo + (hi - lo) / 2;
    sort (aux, a, lo, mid);
    sort (aux, a, mid+1, hi);
    merge(aux, a, lo, mid, hi);
}
```

switch roles of aux[] and a[]
Mergesort: visualization

- first subarray
- second subarray
- first merge
- first half sorted
- second half sorted
- result
Basic plan.

- Pass through array, merging subarrays of size 1.
- Repeat for subarrays of size 2, 4, 8, 16, ....

Bottom line. No recursion needed!
public class MergeBU
{
   private static Comparable[] aux;

   private static void merge(Comparable[] a, int lo, int mid, int hi)
   {
      /* as before */
   }

   public static void sort(Comparable[] a)
   {
      int N = a.length;
      aux = new Comparable[N];
      for (int sz = 1; sz < N; sz = sz+sz)
         for (int lo = 0; lo < N-sz; lo += sz+sz)
            merge(a, lo, lo+sz-1, Math.min(lo+sz+sz-1, N-1));
   }
}
Bottom-up mergesort: visual trace
Sorting Algorithms

- Sorting review
  - Rules of the game
  - Selection sort
  - Insertion sort
  - Shellsort
  - Mergesort
  - Quicksort
- Heapsort
Quicksort

Basic plan.

• **Shuffle** the array.
• **Partition** so that, for some $j$
  - entry $a[j]$ is in place
  - no larger entry to the left of $j$
  - no smaller entry to the right of $j$
• **Sort** each piece recursively.

---

Sir Charles Antony Richard Hoare
1980 Turing Award

**Quicksort overview**
Quicksort partitioning

Repeat until i and j pointers cross.

- Scan i from left to right so long as $a[i] < a[lo]$.
- Scan j from right to left so long as $a[j] > a[lo]$.
- Exchange $a[i]$ with $a[j]$.

Stop i scan because $a[i] \geq a[lo]$
Quicksort partitioning

Repeat until i and j pointers cross.

• Scan i from left to right so long as \(a[i] < a[lo]\).
• Scan j from right to left so long as \(a[j] > a[lo]\).
• Exchange \(a[i]\) with \(a[j]\).
Quicksort partitioning

Repeat until \( i \) and \( j \) pointers cross.

- Scan \( i \) from left to right so long as \( a[i] < a[lo] \).
- Scan \( j \) from right to left so long as \( a[j] > a[lo] \).
- Exchange \( a[i] \) with \( a[j] \).
Quicksort partitioning

Repeat until i and j pointers cross.
- Scan i from left to right so long as \( a[i] < a[lo] \).
- Scan j from right to left so long as \( a[j] > a[lo] \).
- Exchange \( a[i] \) with \( a[j] \).

stop j scan and exchange \( a[i] \) with \( a[j] \)
Quicksort partitioning

Repeat until $i$ and $j$ pointers cross.

- Scan $i$ from left to right so long as $a[i] < a[lo]$.
- Scan $j$ from right to left so long as $a[j] > a[lo]$.
- Exchange $a[i]$ with $a[j]$.
Quicksort partitioning

Repeat until $i$ and $j$ pointers cross.

- Scan $i$ from left to right so long as $a[i] < a[lo]$.
- Scan $j$ from right to left so long as $a[j] > a[lo]$.
- Exchange $a[i]$ with $a[j]$.
Quicksort partitioning

Repeat until \( i \) and \( j \) pointers cross.

- Scan \( i \) from left to right so long as \( a[i] < a[lo] \).
- Scan \( j \) from right to left so long as \( a[j] > a[lo] \).
- Exchange \( a[i] \) with \( a[j] \).

\[
\text{stop } i \text{ scan because } a[i] \geq a[lo]
\]
Quicksort partitioning

Repeat until \( i \) and \( j \) pointers cross.

- Scan \( i \) from left to right so long as \( a[i] < a[lo] \).
- Scan \( j \) from right to left so long as \( a[j] > a[lo] \).
- Exchange \( a[i] \) with \( a[j] \).
Quicksort partitioning

Repeat until $i$ and $j$ pointers cross.

- Scan $i$ from left to right so long as $a[i] < a[lo]$.
- Scan $j$ from right to left so long as $a[j] > a[lo]$.
- Exchange $a[i]$ with $a[j]$.
Quicksort partitioning

Repeat until \( i \) and \( j \) pointers cross.

- Scan \( i \) from left to right so long as \( a[i] < a[lo] \).
- Scan \( j \) from right to left so long as \( a[j] > a[lo] \).
- Exchange \( a[i] \) with \( a[j] \).

\[ K \ C \ A \ T \ E \ L \ E \ P \ U \ I \ M \ Q \ R \ X \ O \ S \]

\[ \begin{array}{c}
\uparrow \\
lo \\
\uparrow \\
i \\
\uparrow \\
j
\end{array} \]

stop \( j \) scan and exchange \( a[i] \) with \( a[j] \)
Quicksort partitioning

Repeat until $i$ and $j$ pointers cross.

- Scan $i$ from left to right so long as $a[i] < a[lo]$.
- Scan $j$ from right to left so long as $a[j] > a[lo]$.
- Exchange $a[i]$ with $a[j]$.
Quicksort partitioning

Repeat until \(i\) and \(j\) pointers cross.
- Scan \(i\) from left to right so long as \(a[i] < a[lo]\).
- Scan \(j\) from right to left so long as \(a[j] > a[lo]\).
- Exchange \(a[i]\) with \(a[j]\).
Quicksort partitioning

Repeat until $i$ and $j$ pointers cross.

- Scan $i$ from left to right so long as $a[i] < a[lo]$.
- Scan $j$ from right to left so long as $a[j] > a[lo]$.
- Exchange $a[i]$ with $a[j]$.

$\uparrow$ $\uparrow$ $\uparrow$

$lo$ $i$ $j$

stop $i$ scan because $a[i] >= a[lo]$
Quicksort partitioning

Repeat until i and j pointers cross.

- Scan i from left to right so long as \(a[i] < a[lo]\).
- Scan j from right to left so long as \(a[j] > a[lo]\).
- Exchange \(a[i]\) with \(a[j]\).
Quicksort partitioning

Repeat until $i$ and $j$ pointers cross.

- Scan $i$ from left to right so long as $a[i] < a[lo]$.
- Scan $j$ from right to left so long as $a[j] > a[lo]$.
- Exchange $a[i]$ with $a[j]$.
Quicksort partitioning

Repeat until i and j pointers cross.

- Scan i from left to right so long as \(a[i] < a[lo]\).
- Scan j from right to left so long as \(a[j] > a[lo]\).
- Exchange \(a[i]\) with \(a[j]\).
Quicksort partitioning

Repeat until $i$ and $j$ pointers cross.

- Scan $i$ from left to right so long as $a[i] < a[lo]$.
- Scan $j$ from right to left so long as $a[j] > a[lo]$.
- Exchange $a[i]$ with $a[j]$.
Quicksort partitioning

Repeat until \(i\) and \(j\) pointers cross.

- Scan \(i\) from left to right so long as \(a[i] < a[lo]\).
- Scan \(j\) from right to left so long as \(a[j] > a[lo]\).
- Exchange \(a[i]\) with \(a[j]\).

\[
\begin{array}{cccccccccccccccc}
K & C & A & I & E & E & L & P & U & T & M & Q & R & X & O & S \\
\uparrow & \uparrow & \uparrow & \uparrow & \uparrow & \uparrow & \uparrow & \uparrow & \uparrow & \uparrow & \uparrow & \uparrow & \uparrow & \uparrow & \uparrow & \uparrow \\
lo & i & j & & & & & & & & & & & & & \\
\end{array}
\]

stop \(i\) scan because \(a[i] \geq a[lo]\)
Quicksort partitioning

Repeat until \( i \) and \( j \) pointers cross.

- Scan \( i \) from left to right so long as \( a[i] < a[lo] \).
- Scan \( j \) from right to left so long as \( a[j] > a[lo] \).
- Exchange \( a[i] \) with \( a[j] \).

\[\begin{array}{cccccccccccccccccc}
K & C & A & I & E & E & L & P & U & T & M & Q & R & X & O & S \\
\uparrow & \uparrow & \uparrow & \uparrow & \uparrow & \uparrow & \uparrow & \uparrow & \uparrow & \uparrow & \uparrow & \uparrow & \uparrow & \uparrow & \uparrow & \uparrow \\
lo & j & i & i & i & i & i & i & i & i & i & i & i & i & i & i \\
\end{array}\]

stop \( j \) scan because \( a[j] \leq a[lo] \)
Quicksort partitioning

Repeat until i and j pointers cross.
• Scan i from left to right so long as \( a[i] < a[lo] \).
• Scan j from right to left so long as \( a[j] > a[lo] \).
• Exchange \( a[i] \) with \( a[j] \).

When pointers cross.
• Exchange \( a[lo] \) with \( a[j] \).

pointers cross: exchange \( a[lo] \) with \( a[j] \)
Quicksort partitioning

Repeat until $i$ and $j$ pointers cross.
- Scan $i$ from left to right so long as $a[i] < a[lo]$.
- Scan $j$ from right to left so long as $a[j] > a[lo]$.
- Exchange $a[i]$ with $a[j]$.

When pointers cross.

partitioned!
Quicksort partitioning

Basic plan.

- Scan $i$ from left for an item that belongs on the right.
- Scan $j$ from right for an item that belongs on the left.
- Exchange $a[i]$ and $a[j]$.
- Repeat until pointers cross.
Quicksort: Java code for partitioning

```java
private static int partition(Comparable[] a, int lo, int hi)
{
    int i = lo, j = hi+1;
    while (true)
    {
        while (less(a[++i], a[lo]))
            if (i == hi) break;

        while (less(a[lo], a[--j]))
            if (j == lo) break;

        if (i >= j) break;
        exch(a, i, j);
    }

    exch(a, lo, j);
    return j;
}
```

Quicksort partitioning overview

before
```
<p>| |</p>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>v</td>
</tr>
</tbody>
</table>
```

<table>
<thead>
<tr>
<th>lo</th>
</tr>
</thead>
</table>

after
```
<p>| |</p>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>v</td>
</tr>
<tr>
<td>v</td>
</tr>
</tbody>
</table>
```

<table>
<thead>
<tr>
<th>hi</th>
</tr>
</thead>
</table>

during
```
| v | ≤ v | ≥ v |
```

<table>
<thead>
<tr>
<th>i</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>j</th>
</tr>
</thead>
</table>
public class Quick
{
    private static int partition(Comparable[] a, int lo, int hi)
    {  /* see previous slide */  }

    public static void sort(Comparable[] a)
    {
        StdRandom.shuffle(a);
        sort(a, 0, a.length - 1);
    }

    private static void sort(Comparable[] a, int lo, int hi)
    {
        if (hi <= lo) return;
        int j = partition(a, lo, hi);
        sort(a, lo, j-1);
        sort(a, j+1, hi);
    }
}

shuffle needed for performance guarantee (stay tuned)
Quicksort trace

<table>
<thead>
<tr>
<th>lo</th>
<th>j</th>
<th>hi</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>5</td>
<td>15</td>
</tr>
<tr>
<td>0</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>0</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>15</td>
</tr>
<tr>
<td>7</td>
<td>9</td>
<td>15</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>13</td>
<td>15</td>
</tr>
<tr>
<td>10</td>
<td>12</td>
<td>12</td>
</tr>
<tr>
<td>10</td>
<td>11</td>
<td>11</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>14</td>
<td>15</td>
</tr>
<tr>
<td>15</td>
<td>15</td>
<td></td>
</tr>
</tbody>
</table>

Quicksort trace (array contents after each partition)

Initial values
random shuffle

no partition
for subarrays
of size 1

result

AC E E I K L M O P Q R S T U X

Quicksort trace
Quicksort animation

50 random items

http://www.sorting-algorithms.com/quick-sort
Quicksort: implementation details

Partitioning in-place. Using an extra array makes partitioning easier (and stable), but is not worth the cost.

Terminating the loop. Testing whether the pointers cross is a bit trickier than it might seem.

Staying in bounds. The \((j == lo)\) test is redundant (why?), but the \((i == hi)\) test is not.

Preserving randomness. Shuffling is needed for performance guarantee.

Equal keys. When duplicates are present, it is (counter-intuitively) better to stop on keys equal to the partitioning item's key.
# Quicksort: empirical analysis

## Running time estimates:
- Home PC executes $10^8$ compares/second.
- Supercomputer executes $10^{12}$ compares/second.

<table>
<thead>
<tr>
<th>computer</th>
<th>insertion sort ($N^2$)</th>
<th>mergesort ($N \log N$)</th>
<th>quicksort ($N \log N$)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>thousand</td>
<td>million</td>
<td>billion</td>
</tr>
<tr>
<td>home</td>
<td>instant</td>
<td>2.8 hours</td>
<td>317 years</td>
</tr>
<tr>
<td>super</td>
<td>instant</td>
<td>1 second</td>
<td>1 week</td>
</tr>
</tbody>
</table>

**Lesson 1.** Good algorithms are better than supercomputers.
**Lesson 2.** Great algorithms are better than good ones.
Quicksort: best-case analysis

Best case. Number of compares is $\sim N \lg N$. 

<table>
<thead>
<tr>
<th>lo</th>
<th>j</th>
<th>hi</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>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>H</td>
<td>A</td>
<td>C</td>
<td>B</td>
<td>F</td>
<td>E</td>
<td>G</td>
<td>D</td>
<td>L</td>
<td>I</td>
<td>K</td>
<td>J</td>
<td>N</td>
<td>M</td>
<td>O</td>
</tr>
<tr>
<td>initial values</td>
<td></td>
<td></td>
<td>H</td>
<td>A</td>
<td>C</td>
<td>B</td>
<td>F</td>
<td>E</td>
<td>G</td>
<td>D</td>
<td>L</td>
<td>I</td>
<td>K</td>
<td>J</td>
<td>N</td>
<td>M</td>
<td>O</td>
</tr>
<tr>
<td>random shuffle</td>
<td>0</td>
<td>7</td>
<td>14</td>
<td>D</td>
<td>A</td>
<td>C</td>
<td>B</td>
<td>F</td>
<td>E</td>
<td>G</td>
<td>H</td>
<td>L</td>
<td>I</td>
<td>K</td>
<td>J</td>
<td>N</td>
<td>M</td>
</tr>
<tr>
<td>0</td>
<td>3</td>
<td>6</td>
<td>B</td>
<td>A</td>
<td>C</td>
<td>D</td>
<td>F</td>
<td>E</td>
<td>G</td>
<td>H</td>
<td>L</td>
<td>I</td>
<td>K</td>
<td>J</td>
<td>N</td>
<td>M</td>
<td>O</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>F</td>
<td>E</td>
<td>G</td>
<td>H</td>
<td>L</td>
<td>I</td>
<td>K</td>
<td>J</td>
<td>N</td>
<td>M</td>
<td>O</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td></td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>F</td>
<td>E</td>
<td>G</td>
<td>H</td>
<td>L</td>
<td>I</td>
<td>K</td>
<td>J</td>
<td>N</td>
<td>M</td>
<td>O</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td></td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>F</td>
<td>E</td>
<td>G</td>
<td>H</td>
<td>L</td>
<td>I</td>
<td>K</td>
<td>J</td>
<td>N</td>
<td>M</td>
<td>O</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>6</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>E</td>
<td>F</td>
<td>G</td>
<td>H</td>
<td>L</td>
<td>I</td>
<td>K</td>
<td>J</td>
<td>N</td>
<td>M</td>
<td>O</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td></td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>E</td>
<td>F</td>
<td>G</td>
<td>H</td>
<td>L</td>
<td>I</td>
<td>K</td>
<td>J</td>
<td>N</td>
<td>M</td>
<td>O</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td></td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>E</td>
<td>F</td>
<td>G</td>
<td>H</td>
<td>L</td>
<td>I</td>
<td>K</td>
<td>J</td>
<td>N</td>
<td>M</td>
<td>O</td>
</tr>
<tr>
<td>8</td>
<td>11</td>
<td>14</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>E</td>
<td>F</td>
<td>G</td>
<td>H</td>
<td>J</td>
<td>I</td>
<td>K</td>
<td>L</td>
<td>N</td>
<td>M</td>
<td>O</td>
</tr>
<tr>
<td>8</td>
<td>9</td>
<td>10</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>E</td>
<td>F</td>
<td>G</td>
<td>H</td>
<td>I</td>
<td>J</td>
<td>K</td>
<td>L</td>
<td>N</td>
<td>M</td>
<td>O</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td></td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>E</td>
<td>F</td>
<td>G</td>
<td>H</td>
<td>I</td>
<td>J</td>
<td>K</td>
<td>L</td>
<td>N</td>
<td>M</td>
<td>O</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td></td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>E</td>
<td>F</td>
<td>G</td>
<td>H</td>
<td>I</td>
<td>J</td>
<td>K</td>
<td>L</td>
<td>N</td>
<td>M</td>
<td>O</td>
</tr>
<tr>
<td>12</td>
<td>13</td>
<td>14</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>E</td>
<td>F</td>
<td>G</td>
<td>H</td>
<td>I</td>
<td>J</td>
<td>K</td>
<td>L</td>
<td>M</td>
<td>N</td>
<td>O</td>
</tr>
<tr>
<td>12</td>
<td>12</td>
<td></td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>E</td>
<td>F</td>
<td>G</td>
<td>H</td>
<td>I</td>
<td>J</td>
<td>K</td>
<td>L</td>
<td>M</td>
<td>N</td>
<td>O</td>
</tr>
<tr>
<td>14</td>
<td>14</td>
<td></td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>E</td>
<td>F</td>
<td>G</td>
<td>H</td>
<td>I</td>
<td>J</td>
<td>K</td>
<td>L</td>
<td>M</td>
<td>N</td>
<td>O</td>
</tr>
<tr>
<td></td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>E</td>
<td>F</td>
<td>G</td>
<td>H</td>
<td>I</td>
<td>J</td>
<td>K</td>
<td>L</td>
<td>M</td>
<td>N</td>
<td>O</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Worst case. Number of compares is $\sim \frac{1}{2} N^2$.

<table>
<thead>
<tr>
<th>lo</th>
<th>j</th>
<th>hi</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>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>E</td>
<td>F</td>
<td>G</td>
<td>H</td>
<td>I</td>
<td>J</td>
<td>K</td>
<td>L</td>
<td>M</td>
<td>N</td>
<td>O</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>E</td>
<td>F</td>
<td>G</td>
<td>H</td>
<td>I</td>
<td>J</td>
<td>K</td>
<td>L</td>
<td>M</td>
<td>N</td>
<td>O</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>E</td>
<td>F</td>
<td>G</td>
<td>H</td>
<td>I</td>
<td>J</td>
<td>K</td>
<td>L</td>
<td>M</td>
<td>N</td>
<td>O</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>E</td>
<td>F</td>
<td>G</td>
<td>H</td>
<td>I</td>
<td>J</td>
<td>K</td>
<td>L</td>
<td>M</td>
<td>N</td>
<td>O</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>E</td>
<td>F</td>
<td>G</td>
<td>H</td>
<td>I</td>
<td>J</td>
<td>K</td>
<td>L</td>
<td>M</td>
<td>N</td>
<td>O</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>E</td>
<td>F</td>
<td>G</td>
<td>H</td>
<td>I</td>
<td>J</td>
<td>K</td>
<td>L</td>
<td>M</td>
<td>N</td>
<td>O</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>E</td>
<td>F</td>
<td>G</td>
<td>H</td>
<td>I</td>
<td>J</td>
<td>K</td>
<td>L</td>
<td>M</td>
<td>N</td>
<td>O</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>E</td>
<td>F</td>
<td>G</td>
<td>H</td>
<td>I</td>
<td>J</td>
<td>K</td>
<td>L</td>
<td>M</td>
<td>N</td>
<td>O</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>E</td>
<td>F</td>
<td>G</td>
<td>H</td>
<td>I</td>
<td>J</td>
<td>K</td>
<td>L</td>
<td>M</td>
<td>N</td>
<td>O</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>E</td>
<td>F</td>
<td>G</td>
<td>H</td>
<td>I</td>
<td>J</td>
<td>K</td>
<td>L</td>
<td>M</td>
<td>N</td>
<td>O</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>E</td>
<td>F</td>
<td>G</td>
<td>H</td>
<td>I</td>
<td>J</td>
<td>K</td>
<td>L</td>
<td>M</td>
<td>N</td>
<td>O</td>
</tr>
</tbody>
</table>

Initial values:

Random shuffle:
**Proposition.** The average number of compares $C_N$ to quicksort an array of $N$ distinct keys is $\sim 2N \ln N$ (and the number of exchanges is $\sim \frac{1}{3} N \ln N$).

**Pf 1.** $C_N$ satisfies the recurrence $C_0 = C_1 = 0$ and for $N \geq 2$:

$$C_N = (N + 1) + \left( \frac{C_0 + C_{N-1}}{N} \right) + \left( \frac{C_1 + C_{N-2}}{N} \right) + \ldots + \left( \frac{C_{N-1} + C_0}{N} \right)$$

- Multiply both sides by $N$ and collect terms:

$$NC_N = N(N + 1) + 2(C_0 + C_1 + \ldots + C_{N-1})$$

- Subtract this from the same equation for $N - 1$:

$$NC_N - (N - 1)C_{N-1} = 2N + 2C_{N-1}$$

- Rearrange terms and divide by $N(N + 1)$:

$$\frac{C_N}{N + 1} = \frac{C_{N-1}}{N} + \frac{2}{N + 1}$$
**Quicksort: average-case analysis**

- Repeatedly apply above equation:

\[
\frac{C_N}{N+1} = \frac{C_{N-1}}{N} + \frac{2}{N+1}
\]

\[
= \frac{C_{N-2}}{N-1} + \frac{2}{N} + \frac{2}{N+1}
\]

\[
= \frac{C_{N-3}}{N-2} + \frac{2}{N-1} + \frac{2}{N} + \frac{2}{N+1}
\]

\[
= \frac{2}{3} + \frac{2}{4} + \frac{2}{5} + \ldots + \frac{2}{N+1}
\]

- Approximate sum by an integral:

\[
C_N = 2(N+1) \left( \frac{1}{3} + \frac{1}{4} + \frac{1}{5} + \ldots + \frac{1}{N+1} \right)
\]

\[
\approx 2(N+1) \int_3^{N+1} \frac{1}{x} \, dx
\]

- Finally, the desired result:

\[
C_N \sim 2(N+1) \ln N \approx 1.39N \lg N
\]
Quicksort: summary of performance characteristics

Worst case. Number of compares is quadratic.
- \[ N + (N - 1) + (N - 2) + \ldots + 1 \sim \frac{1}{2} N^2. \]
- More likely that your computer is struck by lightning bolt.

Average case. Number of compares is \( \sim 1.39 \, N \, \lg N \).
- 39% more compares than mergesort.
- But faster than mergesort in practice because of less data movement.

Random shuffle.
- Probabilistic guarantee against worst case.
- Basis for math model that can be validated with experiments.

Caveat emptor. Many textbook implementations go quadratic if array
- Is sorted or reverse sorted.
- Has many duplicates (even if randomized!)
Insertion sort small subarrays.

- Even quicksort has too much overhead for tiny subarrays.
- Cutoff to insertion sort for $\approx 10$ items.
- Note: could delay insertion sort until one pass at end.

```java
private static void sort(Comparable[] a, int lo, int hi) {
    if (hi <= lo + CUTOFF - 1) {
        Insertion.sort(a, lo, hi);
        return;
    }
    int j = partition(a, lo, hi);
    sort(a, lo, j-1);
    sort(a, j+1, hi);
}
```
**Quicksort: practical improvements**

Median of sample.
- Best choice of pivot item = median.
- Estimate true median by taking median of sample.
- Median-of-3 (random) items.

```
private static void sort(Comparable[] a, int lo, int hi)
{
    if (hi <= lo) return;
    int m = medianOf3(a, lo, lo + (hi - lo)/2, hi);
    swap(a, lo, m);
    int j = partition(a, lo, hi);
    sort(a, lo, j-1);
    sort(a, j+1, hi);
}
```
Quicksort with median-of-3 and cutoff to insertion sort: visualization

input

result of first partition

left subarray partially sorted

both subarrays partially sorted

result
## Sorting summary

<table>
<thead>
<tr>
<th>inplace?</th>
<th>stable?</th>
<th>worst</th>
<th>average</th>
<th>best</th>
<th>remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>selection</strong></td>
<td>✔️</td>
<td>$N^2/2$</td>
<td>$N^2/2$</td>
<td>$N^2/2$</td>
<td>$N$ exchanges</td>
</tr>
<tr>
<td><strong>insertion</strong></td>
<td>✔️</td>
<td>✔️</td>
<td>$N^2/2$</td>
<td>$N^2/4$</td>
<td>$N$ use for small $N$ or partially ordered</td>
</tr>
<tr>
<td><strong>shell</strong></td>
<td>✔️</td>
<td>?</td>
<td>?</td>
<td>$N$</td>
<td>tight code, subquadratic</td>
</tr>
<tr>
<td><strong>merge</strong></td>
<td>✔️</td>
<td>✔️</td>
<td>$N \log N$</td>
<td>$N \log N$</td>
<td>$N \log N$ guarantee, stable</td>
</tr>
<tr>
<td><strong>quick</strong></td>
<td>✔️</td>
<td>✔️</td>
<td>$N^2/2$</td>
<td>$2N \ln N$</td>
<td>$N \log N$ probabilistic guarantee</td>
</tr>
<tr>
<td><strong>3-way quick</strong></td>
<td>✔️</td>
<td>✔️</td>
<td>$N^2/2$</td>
<td>$2N \ln N$</td>
<td>$N$ improves quicksort in presence of duplicate keys</td>
</tr>
<tr>
<td><strong>??</strong></td>
<td>✔️</td>
<td>✔️</td>
<td>$N \log N$</td>
<td>$N \log N$</td>
<td>$N \log N$ holy sorting grail</td>
</tr>
</tbody>
</table>
Sorting Algorithms

- Heapsort
- API
  - Elementary implementations
  - Binary heaps
  - Heapsort
  - Event-driven simulation
Priority queue

Collections. Insert and delete items. Which item to delete?

Stack. Remove the item most recently added.
Queue. Remove the item least recently added.
Randomized queue. Remove a random item.
Priority queue. Remove the largest (or smallest) item.

<table>
<thead>
<tr>
<th>operation</th>
<th>argument</th>
<th>return value</th>
</tr>
</thead>
<tbody>
<tr>
<td>insert</td>
<td>P</td>
<td></td>
</tr>
<tr>
<td>insert</td>
<td>Q</td>
<td></td>
</tr>
<tr>
<td>insert</td>
<td>E</td>
<td></td>
</tr>
<tr>
<td>remove max</td>
<td>X</td>
<td>Q</td>
</tr>
<tr>
<td>insert</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>insert</td>
<td>M</td>
<td></td>
</tr>
<tr>
<td>remove max</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>insert</td>
<td>P</td>
<td></td>
</tr>
<tr>
<td>insert</td>
<td>L</td>
<td></td>
</tr>
<tr>
<td>insert</td>
<td>E</td>
<td></td>
</tr>
<tr>
<td>remove max</td>
<td>P</td>
<td></td>
</tr>
</tbody>
</table>
### Priority queue API

**Requirement.** Generic items are `Comparable`.

```java
public class MaxPQ<Key extends Comparable<Key>>
```

- **MaxPQ()** *(create an empty priority queue)*
- **MaxPQ(Key[] a)** *(create a priority queue with given keys)*
- **void insert(Key v)** *(insert a key into the priority queue)*
- **Key delMax()** *(return and remove the largest key)*
- **boolean isEmpty()** *(is the priority queue empty?)*
- **Key max()** *(return the largest key)*
- **int size()** *(number of entries in the priority queue)*

Key must be Comparable (bounded type parameter)
Priority queue applications

• Event-driven simulation. [customers in a line, colliding particles]
• Numerical computation. [reducing roundoff error]
• Data compression. [Huffman codes]
• Graph searching. [Dijkstra's algorithm, Prim's algorithm]
• Computational number theory. [sum of powers]
• Artificial intelligence. [A* search]
• Statistics. [maintain largest M values in a sequence]
• Operating systems. [load balancing, interrupt handling]
• Discrete optimization. [bin packing, scheduling]
• Spam filtering. [Bayesian spam filter]

Generalizes: stack, queue, randomized queue.
**Challenge.** Find the largest $M$ items in a stream of $N$ items ($N$ huge, $M$ large).

- Fraud detection: isolate $$ transactions.
- File maintenance: find biggest files or directories.

**Constraint.** Not enough memory to store $N$ items.

```plaintext
% more tinyBatch.txt
Turing  6/17/1990   644.08
vonNeumann 3/26/2002 4121.85
Dijkstra  8/22/2007  2678.40
vonNeumann 1/11/1999 4409.74
Dijkstra  11/18/1995  837.42
Hoare    5/10/1993  3229.27
vonNeumann 2/12/1994 4732.35
Hoare    8/18/1992  4381.21
Turing   1/11/2002  66.10
Thompson 2/27/2000  4747.08
vonNeumann 2/12/1994 4732.35
vonNeumann 1/11/1999 4409.74
Hoare    8/18/1992  4381.21
vonNeumann 3/26/2002 4121.85

% java TopM 5 < tinyBatch.txt
Thompson  2/27/2000  4747.08
vonNeumann 2/12/1994 4732.35
vonNeumann 1/11/1999 4409.74
Hoare     8/18/1992  4381.21
vonNeumann 3/26/2002 4121.85
```
Challenge. Find the largest $M$ items in a stream of $N$ items ($N$ huge, $M$ large).

```java
MinPQ<Transaction> pq = new MinPQ<Transaction>();
while (StdIn.hasNextLine())
{
   String line = StdIn.readLine();
   Transaction item = new Transaction(line);
   pq.insert(item);
   if (pq.size() > M)
      pq.delMin();
}
```

Transaction data type is Comparable (ordered by $\leq$)

pq contains largest $M$ items

Use a min-oriented pq

Order of growth of finding the largest $M$ in a stream of $N$ items

<table>
<thead>
<tr>
<th>implementation</th>
<th>time</th>
<th>space</th>
</tr>
</thead>
<tbody>
<tr>
<td>sort</td>
<td>$N \log N$</td>
<td>$N$</td>
</tr>
<tr>
<td>elementary PQ</td>
<td>$M \times N$</td>
<td>$M$</td>
</tr>
<tr>
<td>binary heap</td>
<td>$N \log M$</td>
<td>$M$</td>
</tr>
<tr>
<td>best in theory</td>
<td>$N$</td>
<td>$M$</td>
</tr>
</tbody>
</table>
Sorting Algorithms

- Heapsort
  - API
  - Elementary implementations
  - Binary heaps
  - Heapsort
  - Event-driven simulation
## Priority queue: unordered and ordered array implementation

<table>
<thead>
<tr>
<th>operation</th>
<th>argument</th>
<th>return value</th>
<th>size</th>
<th>contents (unordered)</th>
<th>contents (ordered)</th>
</tr>
</thead>
<tbody>
<tr>
<td>insert</td>
<td>P</td>
<td></td>
<td>1</td>
<td>P</td>
<td>P</td>
</tr>
<tr>
<td>insert</td>
<td>Q</td>
<td></td>
<td>2</td>
<td>P Q</td>
<td>P Q</td>
</tr>
<tr>
<td>insert</td>
<td>E</td>
<td></td>
<td>3</td>
<td>P Q E</td>
<td>E P Q</td>
</tr>
<tr>
<td>remove max</td>
<td>Q</td>
<td></td>
<td>2</td>
<td>P E</td>
<td>E P</td>
</tr>
<tr>
<td>insert</td>
<td>X</td>
<td></td>
<td>3</td>
<td>P E X</td>
<td>E P X</td>
</tr>
<tr>
<td>insert</td>
<td>A</td>
<td></td>
<td>4</td>
<td>P E X A</td>
<td>A E P X</td>
</tr>
<tr>
<td>insert</td>
<td>M</td>
<td></td>
<td>5</td>
<td>P E X A M</td>
<td>A E M P X</td>
</tr>
<tr>
<td>remove max</td>
<td>X</td>
<td></td>
<td>4</td>
<td>P E M A</td>
<td>A E M P</td>
</tr>
<tr>
<td>insert</td>
<td>P</td>
<td></td>
<td>5</td>
<td>P E M A P</td>
<td>A E M P P</td>
</tr>
<tr>
<td>insert</td>
<td>L</td>
<td></td>
<td>6</td>
<td>P E M A P L</td>
<td>A E L M P P</td>
</tr>
<tr>
<td>insert</td>
<td>E</td>
<td></td>
<td>7</td>
<td>P E M A P L E</td>
<td>A E E L M P P</td>
</tr>
<tr>
<td>remove max</td>
<td>P</td>
<td></td>
<td>6</td>
<td>E M A P L E</td>
<td>A E E L M P</td>
</tr>
</tbody>
</table>

A sequence of operations on a priority queue
public class UnorderedMaxPQ<Key extends Comparable<Key>>
{
    private Key[] pq;   // pq[i] = ith element on pq
    private int N;      // number of elements on pq

    public UnorderedMaxPQ(int capacity)
    {  pq = (Key[]) new Comparable[capacity];  }

    public boolean isEmpty()
    {  return N == 0;  }

    public void insert(Key x)
    {  pq[N++] = x;  }

    public Key delMax()
    {
        int max = 0;
        for (int i = 1; i < N; i++)
            if (less(max, i)) max = i;
        exch(max, N-1);
        return pq[--N];
    }
}
Challenge. Implement all operations efficiently.

<table>
<thead>
<tr>
<th>implementation</th>
<th>insert</th>
<th>del max</th>
<th>max</th>
</tr>
</thead>
<tbody>
<tr>
<td>unordered array</td>
<td>1</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>ordered array</td>
<td>N</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>goal</td>
<td>log N</td>
<td>log N</td>
<td>log N</td>
</tr>
</tbody>
</table>

order-of-growth of running time for priority queue with N items
SORTING ALGORITHMS

- Heapsort
- API
- Elementary implementations
- Binary heaps
- Heapsort
- Event-driven simulation
**Binary tree**

**Binary tree.** Empty or node with links to left and right binary trees.

**Complete tree.** Perfectly balanced, except for bottom level.

**Property.** Height of complete tree with \( N \) nodes is \([\lg N]\).

**Pf.** Height only increases when \( N \) is a power of 2.
A complete binary tree in nature

Hyphaene Compressa - Doum Palm
Binary heap representations

**Binary heap.** Array representation of a heap-ordered complete binary tree.

**Heap-ordered binary tree.**
- Keys in nodes.
- Parent's key no smaller than children's keys.

**Array representation.**
- Indices start at 1.
- Take nodes in level order.
- No explicit links needed!
Binary heap properties

Proposition. Largest key is $a[1]$, which is root of binary tree.

Proposition. Can use array indices to move through tree.

- Parent of node at $k$ is at $k/2$.
- Children of node at $k$ are at $2k$ and $2k+1$.
Promotion in a heap

Scenario. Child's key becomes larger key than its parent's key.

To eliminate the violation:
• Exchange key in child with key in parent.
• Repeat until heap order restored.

private void swim(int k)
{
   while (k > 1 && less(k/2, k))
   {
      exch(k, k/2);
      k = k/2;
   }
}

Peter principle. Node promoted to level of incompetence.
**Insertion in a heap**

*Insert.* Add node at end, then swim it up.

*Cost.* At most $1 + \lg N$ compares.

```java
public void insert(Key x) {
    pq[++N] = x;
    swim(N);
}
```
Demotion in a heap

Scenario. Parent's key becomes **smaller** than one (or both) of its children's keys.

To eliminate the violation:
- Exchange key in parent with key in larger child.
- Repeat until heap order restored.

```java
private void sink(int k) {
    while (2*k <= N) {
        int j = 2*k;
        if (j < N && less(j, j+1)) j++;
        if (!less(k, j)) break;
        exch(k, j);
        k = j;
    }
}
```

Power struggle. Better subordinate promoted.
Delete the maximum in a heap

Delete max. Exchange root with node at end, then sink it down.

Cost. At most $2 \lg N$ compares.

```
public Key delMax()
{
   Key max = pq[1];
   exch(1, N--);
   sink(1);
   pq[N+1] = null;
   return max;
}
```
Binary heap operations

**Insert.** Add node at end, then swim it up.

**Remove the maximum.** Exchange root with node at end, then sink it down.

heap ordered
Binary heap operations

**Insert.** Add node at end, then swim it up.

**Remove the maximum.** Exchange root with node at end, then sink it down.

![Binary heap diagram]

- **Insert S**

  - Add node at end, then swim it up.
  - Exchange root with node at end, then sink it down.

```
T P R N H O A E I G
```

- **Add to heap**
Binary heap operations

**Insert.** Add node at end, then swim it up.

**Remove the maximum.** Exchange root with node at end, then sink it down.

**insert S**

```
T
P
N  H
E  I  G  S
O  R  A
```

violates heap order (swim up)
Binary heap operations

**Insert.** Add node at end, then swim it up.

**Remove the maximum.** Exchange root with node at end, then sink it down.

**insert S**
**Binary heap operations**

**Insert.** Add node at end, then swim it up.

**Remove the maximum.** Exchange root with node at end, then sink it down.

**insert S**

---

**Diagram:**

- **Node S** is added at the end and needs to be swum up.
- The maximum is removed by exchanging the root with the node at the end and then sinking it down.
- The diagram shows the heap after the operations, with nodes labeled with their values and the heap property verified.

---

**Table:**

<table>
<thead>
<tr>
<th></th>
<th>T</th>
<th>S</th>
<th>R</th>
<th>N</th>
<th>P</th>
<th>O</th>
<th>A</th>
<th>E</th>
<th>I</th>
<th>G</th>
<th>H</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2</td>
<td>5</td>
<td></td>
<td></td>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---
**Binary heap operations**

**Insert.** Add node at end, then swim it up.

**Remove the maximum.** Exchange root with node at end, then sink it down.

heap ordered
Binary heap operations

**Insert.** Add node at end, then swim it up.
**Remove the maximum.** Exchange root with node at end, then sink it down.

remove the maximum
**Binary heap operations**

**Insert.** Add node at end, then swim it up.

**Remove the maximum.** Exchange root with node at end, then sink it down.

remove the maximum

![Binary heap diagram](image-url)
Binary heap operations

Insert. Add node at end, then swim it up.
Remove the maximum. Exchange root with node at end, then sink it down.
Binary heap operations

**Insert.** Add node at end, then swim it up.

**Remove the maximum.** Exchange root with node at end, then sink it down.

remove the maximum

```
Insert: Add node at end, then swim it up.
Remove the maximum: Exchange root with node at end, then sink it down.
```
Binary heap operations

**Insert.** Add node at end, then swim it up.

**Remove the maximum.** Exchange root with node at end, then sink it down.

Binary heap operations

**Insert.** Add node at end, then swim it up.

**Remove the maximum.** Exchange root with node at end, then sink it down.

**remove the maximum**

![Binary heap diagram]

- **S** removes the maximum and violates heap order, so it is sunk down.
**Binary heap operations**

**Insert.** Add node at end, then swim it up.

**Remove the maximum.** Exchange root with node at end, then sink it down.

**heap ordered**
Binary heap operations

**Insert.** Add node at end, then swim it up.

**Remove the maximum.** Exchange root with node at end, then sink it down.
Binary heap operations

Insert. Add node at end, then swim it up.

Remove the maximum. Exchange root with node at end, then sink it down.

remove the maximum

exchange with root
**Binary heap operations**

**Insert.** Add node at end, then swim it up.

**Remove the maximum.** Exchange root with node at end, then sink it down.

---

**remove the maximum**

---

![Binary heap diagram](image-url)
Binary heap operations

**Insert.** Add node at end, then swim it up.

**Remove the maximum.** Exchange root with node at end, then sink it down.

Remove the maximum

---

Diagram of a binary heap with nodes labeled P, N, H, R, O, A, E, I, S. The root node is labeled G. The diagram illustrates the process of removing the maximum value, which involves swapping the root (G) with the node at the end (S) and then sinking the new root down to maintain heap order.

---

G | P | R | N | H | O | A | E | I | S

1
Binary heap operations

**Insert.** Add node at end, then swim it up.

**Remove the maximum.** Exchange root with node at end, then sink it down.

---

**remove the maximum**

---

![Binary heap diagram](image-url)

- **R**
- **P**
- **G**
- **N**
- **H**
- **O**
- **A**
- **E**
- **I**

---

**Array representation:**

<table>
<thead>
<tr>
<th>R</th>
<th>P</th>
<th>G</th>
<th>N</th>
<th>H</th>
<th>O</th>
<th>A</th>
<th>E</th>
<th>I</th>
<th>S</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Binary heap operations

**Insert.** Add node at end, then swim it up.

**Remove the maximum.** Exchange root with node at end, then sink it down.

*remove the maximum*
Binary heap operations

**Insert.** Add node at end, then swim it up.

**Remove the maximum.** Exchange root with node at end, then sink it down.

heap ordered
**Binary heap operations**

**Insert.** Add node at end, then swim it up.

**Remove the maximum.** Exchange root with node at end, then sink it down.

**insert S**
**Binary heap operations**

- **Insert.** Add node at end, then swim it up.
- **Remove the maximum.** Exchange root with node at end, then sink it down.

**insert S**

![Binary heap diagram](image-url)

- The heap order is violated by the addition of S.
- S is swum up to maintain the heap order.
**Binary heap operations**

**Insert.** Add node at end, then swim it up.

**Remove the maximum.** Exchange root with node at end, then sink it down.

**insert S**

![Binary heap diagram]

- **R**
- **P**
- **N**
- **E**
- **I**
- **H**
- **S**
- **G**
- **O**
- **A**

- **violates heap order (swim up)**

<table>
<thead>
<tr>
<th>R</th>
<th>P</th>
<th>O</th>
<th>N</th>
<th>S</th>
<th>G</th>
<th>A</th>
<th>E</th>
<th>I</th>
<th>H</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

233
Binary heap operations

**Insert.** Add node at end, then swim it up.

**Remove the maximum.** Exchange root with node at end, then sink it down.

```
insert S
```

```
E  I  H  P  S
N
```

```
2  5  10
```

violates heap order (swim up)
**Binary heap operations**

**Insert.** Add node at end, then swim it up.

**Remove the maximum.** Exchange root with node at end, then sink it down.

**insert S**

![Binary heap diagram with example insert and violation.](diagram.png)
**Binary heap operations**

**Insert.** Add node at end, then swim it up.

**Remove the maximum.** Exchange root with node at end, then sink it down.

**heap ordered**

```
  S
 /   \
R     O
 |     |
N     G
|     |
E     A
|     |
I     H
```
Binary heap: Java implementation

```java
public class MaxPQ<Key extends Comparable<Key>>
{
   private Key[] pq;
   private int N;

   public MaxPQ(int capacity)
   {   pq = (Key[]) new Comparable[capacity+1];   }

   public boolean isEmpty()
   {   return N == 0;   }

   public void insert(Key key)
   {   /* see previous code */   }

   public Key delMax()
   {   /* see previous code */   }

   private void swim(int k)
   {   /* see previous code */   }

   private void sink(int k)
   {   /* see previous code */   }

   private boolean less(int i, int j)
   {   return pq[i].compareTo(pq[j]) < 0;   }

   private void exch(int i, int j)
   {   Key t = pq[i]; pq[i] = pq[j]; pq[j] = t;   }
}
```
Priority queues implementation cost summary

<table>
<thead>
<tr>
<th>implementation</th>
<th>insert</th>
<th>del max</th>
<th>max</th>
</tr>
</thead>
<tbody>
<tr>
<td>unordered array</td>
<td>1</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>ordered array</td>
<td>N</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>binary heap</td>
<td>log N</td>
<td>log N</td>
<td>1</td>
</tr>
<tr>
<td>d-ary heap</td>
<td>logd N</td>
<td>d logd N</td>
<td>1</td>
</tr>
<tr>
<td>Fibonacci</td>
<td>1</td>
<td>log N †</td>
<td>1</td>
</tr>
<tr>
<td>impossible</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

† amortized

order-of-growth of running time for priority queue with N items

why impossible?
Binary heap considerations

Immutability of keys.
• Assumption: client does not change keys while they're on the PQ.
• Best practice: use immutable keys.

Underflow and overflow.
• Underflow: throw exception if deleting from empty PQ.
• Overflow: add no-arg constructor and use resizing array.

Minimum-oriented priority queue.
• Replace less() with greater().
• Implement greater().

Other operations.
• Remove an arbitrary item.
• Change the priority of an item.
Immutability: implementing in Java

Data type. Set of values and operations on those values. Immutable data type. Can't change the data type value once created.

```java
public final class Vector {
    private final int N;
    private final double[] data;

    public Vector(double[] data) {
        this.N = data.length;
        this.data = new double[N];
        for (int i = 0; i < N; i++)
            this.data[i] = data[i];
    }
    ...
}
```

Immutability: properties

Data type. Set of values and operations on those values.
Immutable data type. Can't change the data type value once created.

Advantages.
• Simplifies debugging.
• Safer in presence of hostile code.
• Simplifies concurrent programming.
• Safe to use as key in priority queue or symbol table.

Disadvantage. Must create new object for each data type value.

“Classes should be immutable unless there's a very good reason to make them mutable.... If a class cannot be made immutable, you should still limit its mutability as much as possible.”
— Joshua Bloch (Java architect)
SORTING ALGORITHMS

- Heapsort
- API
- Elementary implementations
- Binary heaps
- Heapsort
- Event-driven simulation
Heapsort

Basic plan for in-place sort.

- Create max-heap with all $N$ keys.
- Repeatedly remove the maximum key.
Starting point. Array in arbitrary order.

we assume array entries are indexed 1 to N
Heapsort

Heap construction. Build max heap using bottom-up method.
Heap construction. Build max heap using bottom-up method.

sink 5
Heap construction. Build max heap using bottom-up method.

sink 5
Heapsort

Heap construction. Build max heap using bottom-up method.

sink 5

3-node heap
Heap construction. Build max heap using bottom-up method.
Heap construction. Build max heap using bottom-up method.

sink 4
Heap construction. Build max heap using bottom-up method.

sink 3
Heapsort

Heap construction. Build max heap using bottom-up method.

sink 3
Heap construction. Build max heap using bottom-up method.

sink 3

3-node heap
Heap sort

Heap construction. Build max heap using bottom-up method.

sink 2
Heap construction. Build max heap using bottom-up method.

sink 2
Heap construction. Build max heap using bottom-up method.

sink 2
Heap construction. Build max heap using bottom-up method.

sink 2

7-node heap
Heap construction. Build max heap using bottom-up method.

sink 1
Heap construction. Build max heap using bottom-up method.

sink 1
Heapsort

Heap construction. Build max heap using bottom-up method.

end of construction phase

11-node heap

X

T

P

M

O

E

L

E

S

R

A

X

T

S

P

L

R

A

M

O

E

E
Heapsort

Sortdown. Repeatedly delete the largest remaining item.

exchange 1 and 11

Heapsort

Sortdown. Repeatedly delete the largest remaining item.

exchange 1 and 11

```plaintext
E T S P L R A M O E X
1 1
```
**Heapsort**

**Sortdown.** Repeatedly delete the largest remaining item.

sink 1
Heapsort

Sortdown. Repeatedly delete the largest remaining item.

sink 1
Heapsort

Sortdown. Repeatedly delete the largest remaining item.
Heapsort

Sortdown. Repeatedly delete the largest remaining item.
Heapsort

**Sortdown.** Repeatedly delete the largest remaining item.
Sortdown. Repeatedly delete the largest remaining item.

exchange 1 and 10
Heapsort

**Sortdown.** Repeatedly delete the largest remaining item.

exchange 1 and 10

```
1 10

E  P  S  O  L  R  A  M  E  T  X
1  10
```
Heapsort

**Sortdown.** Repeatedly delete the largest remaining item.

sink 1

![Heap diagram with nodes labeled E, P, O, M, L, E, S, R, A, T, X. The root node E is at the top, and the nodes are arranged in a binary tree structure.](image-url)
Heapsort

Sortdown. Repeatedly delete the largest remaining item.

sink 1
Heapsort

Sortdown. Repeatedly delete the largest remaining item.

sink 1
Heapsort

Sortdown. Repeatedly delete the largest remaining item.
Heapsort

Sortdown. Repeatedly delete the largest remaining item.

exchange 1 and 9
Heapsort

**Sortdown.** Repeatedly delete the largest remaining item.

exchange 1 and 9
Heapsort

Sortdown. Repeatedly delete the largest remaining item.

sink 1

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

```
M  S  T  X
```
Heapsort

Sortdown. Repeatedly delete the largest remaining item.

sink 1

sink 1
Heapsort

Sortdown. Repeatedly delete the largest remaining item.
Heapsort

Sortdown. Repeatedly delete the largest remaining item.

exchange 1 and 8
Heapsort

Sortdown. Repeatedly delete the largest remaining item.

exchange 1 and 8
Heapsort

**Sortdown.** Repeatedly delete the largest remaining item.

sink 1
Heapsort

Sortdown. Repeatedly delete the largest remaining item.

sink 1
Heapsort

Sortdown. Repeatedly delete the largest remaining item.

sink 1
Heapsort

Sortdown. Repeatedly delete the largest remaining item.
Heapsort

Sortdown. Repeatedly delete the largest remaining item.

exchange 1 and 7

R S T X

P O E M L E A R S T X

1 7
Sortdown. Repeatedly delete the largest remaining item.

exchange 1 and 7
**Heapsort**

Sortdown. Repeatedly delete the largest remaining item.

sink 1
Heapsort

Sortdown. Repeatedly delete the largest remaining item.

sink 1
**Heapsort**

**Sortdown.** Repeatedly delete the largest remaining item.
**Heapsort**

**Sortdown.** Repeatedly delete the largest remaining item.

sink 1
Heapsort

**Sortdown.** Repeatedly delete the largest remaining item.

exchange 1 and 6

```
  O     M     E
 /     /     |
A     L     E
```

1       6
P       X
R       S
T       T
X       X
Heapsort

**Sortdown.** Repeatedly delete the largest remaining item.

exchange 1 and 6
**Heapsort**

**Sortdown.** Repeatedly delete the largest remaining item.
Heapsort

Sortdown. Repeatedly delete the largest remaining item.
Heapsort

**Sortdown.** Repeatedly delete the largest remaining item.

sink 1
Heapsort

**Sortdown.** Repeatedly delete the largest remaining item.
**Heapsort**

Sortdown. Repeatedly delete the largest remaining item.

exchange 1 and 5
Heapsort

Sortdown. Repeatedly delete the largest remaining item.

exchange 1 and 5
Heapsort

**Sortdown.** Repeatedly delete the largest remaining item.

sink 1
Heapsort

Sortdown. Repeatedly delete the largest remaining item.
Sortdown. Repeatedly delete the largest remaining item.
Heapsort

**Sortdown.** Repeatedly delete the largest remaining item.

exchange 1 and 4
Heapsort

Sortdown. Repeatedly delete the largest remaining item.

exchange 1 and 4
**Heapsort**

**Sortdown.** Repeatedly delete the largest remaining item.
Heapsort

**Sortdown.** Repeatedly delete the largest remaining item.

sink 1
**Heapsort**

**Sortdown.** Repeatedly delete the largest remaining item.
Heapsort

Sortdown. Repeatedly delete the largest remaining item.

exchange 1 and 3
Heapsort

Sortdown. Repeatedly delete the largest remaining item.

exchange 1 and 3
**Heapsort**

**Sortdown.** Repeatedly delete the largest remaining item.

-sink 1-
**Heapsort**

**Sortdown.** Repeatedly delete the largest remaining item.
Heapsort

**Sortdown.** Repeatedly delete the largest remaining item.

exchange 1 and 2
Heapsort

Sortdown. Repeatedly delete the largest remaining item.

exchange 1 and 2

A

E

L

M

O

P

R

S

T

X

1 2
Sortdown. Repeatedly delete the largest remaining item.
Heapsort

Sortdown. Repeatedly delete the largest remaining item.

end of sortdown phase
Heapsort

Ending point. Array in sorted order.
Heapsort: heap construction

First pass. Build heap using bottom-up method.

for (int k = N/2; k >= 1; k--)
   sink(a, k, N);
Heapsort: sortdown

Second pass.

- Remove the maximum, one at a time.
- Leave in array, instead of nulling out.

```java
while (N > 1)
{
    exch(a, 1, N--);
    sink(a, 1, N);
}
```
Heapsort: Java implementation

```java
public class Heap {
    public static void sort(Comparable[] pq) {
        int N = pq.length;
        for (int k = N/2; k >= 1; k--)
            sink(pq, k, N);
        while (N > 1) {
            exch(pq, 1, N);
            sink(pq, 1, --N);
        }
    }

    private static void sink(Comparable[] pq, int k, int N) {
        /* as before */
    }

    private static boolean less(Comparable[] pq, int i, int j) {
        /* as before */
    }

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

but convert from 1-based indexing to 0-base indexing
Heapsort: trace

<table>
<thead>
<tr>
<th>N</th>
<th>k</th>
<th>a[i]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td><strong>initial values</strong></td>
</tr>
<tr>
<td>11</td>
<td>5</td>
<td>SORT LXAMPLE E</td>
</tr>
<tr>
<td>11</td>
<td>4</td>
<td>SORT LXAMPLE E</td>
</tr>
<tr>
<td>11</td>
<td>3</td>
<td>SORT LXAMPLE E</td>
</tr>
<tr>
<td>11</td>
<td>2</td>
<td>SXLPRAMOE E</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>XTSPLRAMOE E</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>sorted result</strong></td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>XTSPLRAMOE E</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>XTSPLRAMOE E</td>
</tr>
<tr>
<td>9</td>
<td>1</td>
<td>XTSPLRAMOE E</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>XTSPLRAMOE E</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>XTSPLRAMOE E</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>XTSPLRAMOE E</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>XTSPLRAMOE E</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>XTSPLRAMOE E</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>XTSPLRAMOE E</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>XTSPLRAMOE E</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>XTSPLRAMOE E</td>
</tr>
</tbody>
</table>

Heapsort trace (array contents just after each sink)
Heapsort animation

50 random items

http://www.sorting-algorithms.com/heap-sort
Heapsort: mathematical analysis

Proposition. Heap construction uses fewer than $2N$ compares and exchanges.

Proposition. Heapsort uses at most $2N \lg N$ compares and exchanges.

Significance. In-place sorting algorithm with $N \log N$ worst-case.

- Mergesort: no, linear extra space.  
  in-place merge possible, not practical
- Quicksort: no, quadratic time in worst case.  
  N log N worst-case quicksort possible, not practical
- Heapsort: yes!

Bottom line. Heapsort is optimal for both time and space, but:

- Inner loop longer than quicksort’s.
- Makes poor use of cache memory.
- Not stable.
# Sorting algorithms: summary

<table>
<thead>
<tr>
<th>inplace?</th>
<th>stable?</th>
<th>worst</th>
<th>average</th>
<th>best</th>
<th>remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>selection</td>
<td>x</td>
<td>$N^2 / 2$</td>
<td>$N^2 / 2$</td>
<td>$N^2 / 2$</td>
<td>N exchanges</td>
</tr>
<tr>
<td>insertion</td>
<td>x</td>
<td>x</td>
<td>$N^2 / 2$</td>
<td>$N^2 / 4$</td>
<td>N</td>
</tr>
<tr>
<td>shell</td>
<td>x</td>
<td>?</td>
<td>?</td>
<td>N</td>
<td>tight code, subquadratic</td>
</tr>
<tr>
<td>quick</td>
<td>x</td>
<td>$N^2 / 2$</td>
<td>$2N \ln N$</td>
<td>$N \lg N$</td>
<td>$N \log N$ probabilistic guarantee fastest in practice</td>
</tr>
<tr>
<td>3-way quick</td>
<td>x</td>
<td>$N^2 / 2$</td>
<td>$2N \ln N$</td>
<td>N</td>
<td>improves quicksort in presence of duplicate keys</td>
</tr>
<tr>
<td>merge</td>
<td>x</td>
<td>$N \lg N$</td>
<td>$N \lg N$</td>
<td>$N \lg N$</td>
<td>$N \log N$ guarantee, stable</td>
</tr>
<tr>
<td>heap</td>
<td>x</td>
<td>$2N \lg N$</td>
<td>$2N \lg N$</td>
<td>$N \lg N$</td>
<td>$N \log N$ guarantee, in-place</td>
</tr>
<tr>
<td>???</td>
<td>x</td>
<td>x</td>
<td>$N \lg N$</td>
<td>$N \lg N$</td>
<td>$N \log N$ holy sorting grail</td>
</tr>
</tbody>
</table>
SORTING ALGORITHMS

- Heapsort
  - API
  - Elementary implementations
  - Binary heaps
  - Heapsort
  - Event-driven simulation
Goal. Simulate the motion of $N$ moving particles that behave according to the laws of elastic collision.
Molecular dynamics simulation of hard discs

**Goal.** Simulate the motion of $N$ moving particles that behave according to the laws of elastic collision.

**Hard disc model.**
- Moving particles interact via elastic collisions with each other and walls.
- Each particle is a disc with known position, velocity, mass, and radius.
- No other forces.

**Significance.** Relates macroscopic observables to microscopic dynamics.
- Einstein: explain Brownian motion of pollen grains.
Warmup: bouncing balls

Time-driven simulation. $N$ bouncing balls in the unit square.

```java
public class BouncingBalls
{
   public static void main(String[] args)
   {
      int N = Integer.parseInt(args[0]);
      Ball[] balls = new Ball[N];
      for (int i = 0; i < N; i++)
         balls[i] = new Ball();
      while(true)
      {
         StdDraw.clear();
         for (int i = 0; i < N; i++)
            {
               balls[i].move(0.5);
               balls[i].draw();
            }
         StdDraw.show(50);
      }
   }
}
```

% java BouncingBalls 100
Warmup: bouncing balls

public class Ball
{
    private double rx, ry;        // position
    private double vx, vy;        // velocity
    private final double radius;  // radius

    public Ball()
    {
        /* initialize position and velocity */
    }

    public void move(double dt)
    {
        if ((rx + vx*dt < radius) || (rx + vx*dt > 1.0 - radius)) { vx = -vx; }
        if ((ry + vy*dt < radius) || (ry + vy*dt > 1.0 - radius)) { vy = -vy; }
        rx = rx + vx*dt;
        ry = ry + vy*dt;
    }

    public void draw()
    {  StdDraw.filledCircle(rx, ry, radius);  }
}

check for collision with walls

Missing. Check for balls colliding with each other.
• Physics problems: when? what effect?
• CS problems: which object does the check? too many checks?
Time-driven simulation

- Discretize time in quanta of size $dt$.
- Update the position of each particle after every $dt$ units of time, and check for overlaps.
- If overlap, roll back the clock to the time of the collision, update the velocities of the colliding particles, and continue the simulation.

![Diagram of time-driven simulation](image)
Main drawbacks.

• $\sim N^2/2$ overlap checks per time quantum.
• Simulation is too slow if $dt$ is very small.
• May miss collisions if $dt$ is too large.
  (if colliding particles fail to overlap when we are looking)
Event-driven simulation

Change state only when something happens.
- Between collisions, particles move in straight-line trajectories.
- Focus only on times when collisions occur.
- Maintain PQ of collision events, prioritized by time.
- Remove the min = get next collision.

Collision prediction. Given position, velocity, and radius of a particle, when will it collide next with a wall or another particle?

Collision resolution. If collision occurs, update colliding particle(s) according to laws of elastic collisions.
Particle-wall collision

Collision prediction and resolution.
- Particle of radius $s$ at position $(rx, ry)$.
- Particle moving in unit box with velocity $(vx, vy)$.
- Will it collide with a vertical wall? If so, when?

**Prediction (at time $t$)**

\[ dt = \text{time to hit wall} = \frac{\text{distance/velocity}}{v_x} = \frac{(1 - s - r_x)}{v_x} \]

**Resolution (at time $t + dt$)**

- velocity after collision = $(-v_x, vy)$
- position after collision = $(1 - s, r_y + vy dt)$

Predicting and resolving a particle-wall collision
Particle-particle collision prediction

Collision prediction.

- Particle $i$: radius $s_i$, position $(rx_i, ry_i)$, velocity $(vx_i, vy_i)$.
- Particle $j$: radius $s_j$, position $(rx_j, ry_j)$, velocity $(vx_j, vy_j)$.
- Will particles $i$ and $j$ collide? If so, when?
Particle-particle collision prediction

Collision prediction.

- Particle \(i\): radius \(s_i\), position \((rx_i, ry_i)\), velocity \((vx_i, vy_i)\).
- Particle \(j\): radius \(s_j\), position \((rx_j, ry_j)\), velocity \((vx_j, vy_j)\).
- Will particles \(i\) and \(j\) collide? If so, when?

\[
\Delta t = \begin{cases} 
\infty & \text{if } \Delta v \cdot \Delta r \geq 0 \\
\infty & \text{if } d < 0 \\
- \frac{\Delta v \cdot \Delta r + \sqrt{d}}{\Delta v \cdot \Delta v} & \text{otherwise}
\end{cases}
\]

\[
d = (\Delta v \cdot \Delta r)^2 - (\Delta v \cdot \Delta v) (\Delta r \cdot \Delta r - \sigma^2)
\]

\[
\sigma = \sigma_i + \sigma_j
\]

\[
\Delta v = (\Delta vx, \Delta vy) = (vx_i - vx_j, vy_i - vy_j)
\]

\[
\Delta r = (\Delta rx, \Delta ry) = (rx_i - rx_j, ry_i - ry_j)
\]

\[
\Delta v \cdot \Delta v = (\Delta vx)^2 + (\Delta vy)^2
\]

\[
\Delta r \cdot \Delta r = (\Delta rx)^2 + (\Delta ry)^2
\]

Important note: This is high-school physics, so we won't be testing you on it!
Collision resolution. When two particles collide, how does velocity change?

\[
\begin{align*}
    vx'_i &= vx_i + Jx / m_i \\
    vy'_i &= vy_i + Jy / m_i \\
    vx'_j &= vx_j - Jx / m_j \\
    vy'_j &= vy_j - Jy / m_j
\end{align*}
\]

Newton's second law (momentum form)

\[
Jx = \frac{J \Delta rx}{\sigma}, \quad Jy = \frac{J \Delta ry}{\sigma}, \quad J = \frac{2m_i m_j (\Delta v \cdot \Delta r)}{\sigma (m_i + m_j)}
\]

Impulse due to normal force
(conservation of energy, conservation of momentum)

Important note: This is high-school physics, so we won't be testing you on it!
public class Particle
{
    private double rx, ry;       // position
    private double vx, vy;       // velocity
    private final double radius; // radius
    private final double mass;   // mass
    private int count;           // number of collisions

    public Particle(...) { }

    public void move(double dt) { }
    public void draw()          { }

    public double timeToHit(Particle that) { }
    public double timeToHitVerticalWall()   { }
    public double timeToHitHorizontalWall() { }

    public void bounceOff(Particle that)   { }
    public void bounceOffVerticalWall()     { }
    public void bounceOffHorizontalWall()   { }
}

predict collision with particle or wall
resolve collision with particle or wall
Particle-particle collision and resolution

```java
public double timeToHit(Particle that)
{
    if (this == that) return INFINITY;
    double dx  = that.rx - this.rx, dy  = that.ry - this.ry;
    double dvx = that.vx - this.vx; dvy = that.vy - this.vy;
    double dvdr = dx*dvx + dy*dvy;
    if( dvdr > 0) return INFINITY;
    double dvdv = dvx*dvx + dvy*dvy;
    double drdr = dx*dx + dy*dy;
    double sigma = this.radius + that.radius;
    double d = (dvdr*dvdr) - dvdv * (drdr - sigma*sigma);
    if (d < 0) return INFINITY;
    return -(dvdr + Math.sqrt(d)) / dvdv;
}
```

```java
public void bounceOff(Particle that)
{
    double dx  = that.rx - this.rx, dy  = that.ry - this.ry;
    double dvx = that.vx - this.vx, dvy = that.vy - this.vy;
    double dvdr = dx*dvx + dy*dvy;
    double dist = this.radius + that.radius;
    double J = 2 * this.mass * that.mass * dvdr / ((this.mass + that.mass) * dist);
    double Jx = J * dx / dist;
    double Jy = J * dy / dist;
    this.vx += Jx / this.mass;
    this.vy += Jy / this.mass;
    that.vx -= Jx / that.mass;
    that.vy -= Jy / that.mass;
    this.count++;
    that.count++;
}
```

Important note: This is high-school physics, so we won’t be testing you on it!
Collision system: event-driven simulation main

Initialization.
• Fill PQ with all potential particle-wall collisions.
• Fill PQ with all potential particle-particle collisions.

Main loop.
• Delete the impending event from PQ (min priority = $t$).
• If the event has been invalidated, ignore it.
• Advance all particles to time $t$, on a straight-line trajectory.
• Update the velocities of the colliding particle(s).
• Predict future particle-wall and particle-particle collisions involving the colliding particle(s) and insert events onto PQ.

“potential” since collision may not happen if some other collision intervenes
Conventions.

- Neither particle null ⇒ particle-particle collision.
- One particle null ⇒ particle-wall collision.
- Both particles null ⇒ redraw event.

```java
private class Event implements Comparable<Event>
{
    private double time;         // time of event
    private Particle a, b;       // particles involved in event
    private int countA, countB;  // collision counts for a and b

    public Event(double t, Particle a, Particle b) { }

    public int compareTo(Event that) {   return this.time - that.time;   }

    public boolean isValid() {   }
}
```

create event

ordered by time

invalid if intervening collision
public class CollisionSystem
{
    private MinPQ<Event> pq;        // the priority queue
    private double t  = 0.0;  // simulation clock time
    private Particle[] particles; // the array of particles

    public CollisionSystem(Particle[] particles) { }

    private void predict(Particle a) {
        if (a == null) return;
        for (int i = 0; i < N; i++) {
            double dt = a.timeToHit(particles[i]);
            pq.insert(new Event(t + dt, a, particles[i]));
        }
        pq.insert(new Event(t + a.timeToHitVerticalWall(), null, a));
        pq.insert(new Event(t + a.timeToHitHorizontalWall(), null, a));
    }

    private void redraw() { }
    public void simulate() { /* see next slide */ }
}
public void simulate()
{
    pq = new MinPQ<Event>();
    for(int i = 0; i < N; i++) predict(particles[i]);
    pq.insert(new Event(0, null, null));

    while(!pq.isEmpty())
    {
        Event event = pq.delMin();
        if(!event.isValid()) continue;
        Particle a = event.a;
        Particle b = event.b;

        for(int i = 0; i < N; i++)
            particles[i].move(event.time - t);
        t = event.time;

        if      (a != null && b != null) a.bounceOff(b);
        else if (a != null && b == null) a.bounceOffVerticalWall();
        else if (a == null && b != null) b.bounceOffHorizontalWall();
        else if (a == null && b == null) redraw();

        predict(a);
        predict(b);
    }
}
Particle collision simulation example 1

% java CollisionSystem 100
Particle collision simulation example 2

% java CollisionSystem < billiards.txt
Particle collision simulation example 3

% java CollisionSystem < brownian.txt
Particle collision simulation example 4

% java CollisionSystem < diffusion.txt