2.2 **MERGESORT**

- mergesort
- *bottom-up mergesort*
- *sorting complexity*
- *divide-and-conquer*
Two classic sorting algorithms: mergesort and quicksort

Critical components in the world’s computational infrastructure.
- Full scientific understanding of their properties has enabled us to develop them into practical system sorts.
- Quicksort honored as one of top 10 algorithms of 20th century in science and engineering.

Mergesort. [today]

Quicksort. [next lecture]
2.2 Mergesort

- mergesort
- bottom-up mergesort
- sorting complexity
- divide-and-conquer
**Mergesort**

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

---

**Mergesort overview**

```
input  M E R G E S O R T E X A M P L E
sort left half  E E G M O R R S | T E X A M P L E
sort right half  E E G M O R R S   A E E L M P T X
merge results  A E E E E G L M M O P R R S T X
```
Merging demo

**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>lo</th>
<th>mid</th>
<th>mid+1</th>
<th>hi</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>E</td>
<td>G</td>
<td>M</td>
</tr>
<tr>
<td>R</td>
<td>A</td>
<td>C</td>
<td>E</td>
</tr>
<tr>
<td>R</td>
<td>E</td>
<td>R</td>
<td>T</td>
</tr>
</tbody>
</table>

_copy to auxiliary array_

| aux[] |
**Merging demo**

**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>
</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>
</table>
Merging demo

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

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

\[
k
\]

compare minimum in each subarray

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

\[
i & j
\]
Merging demo

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

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

**compare minimum in each subarray**

\[
\begin{array}{cccccccccccc}
\text{aux[]} & & E & E & G & M & R & A & C & E & R & T \\
\text{i} & & & & & & & & & & & \\
\text{j} & & & & & & & & & & & \\
\end{array}
\]
**Goal.** Given two sorted subarrays \(a[lo]\) to \(a[mid]\) and \(a[mid+1]\) to \(a[hi]\), replace with sorted subarray \(a[lo]\) to \(a[hi]\).

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

**Compare minimum in each subarray**

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

---

**Merging demo**

```plaintext
\begin{align*}
\text{a[]} & \quad \begin{array}{cccccccc}
A & C & G & M & R & A & C & E & R & T \\
\end{array} \\
\text{k} & \\
\text{aux[]} & \quad \begin{array}{cccccccc}
E & E & G & M & R & A & C & E & R & T \\
\end{array} \\
\text{i} & \quad \text{j}
\end{align*}
```

**compare minimum in each subarray**
**Merging demo**

**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>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>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>
<td></td>
</tr>
<tr>
<td>$j$</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>
<td></td>
</tr>
</tbody>
</table>
**Merging demo**

**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 with arrays and comparison points](image)

**compare minimum in each subarray**

![Auxiliary array](image)
**Merging demo**

**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>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>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>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>j</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
**Merging demo**

**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>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>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</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>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</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>
```
**Merging demo**

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

```
\[ a[] = \begin{array}{cccccccc}
  & A & C & E & E & R & A & C & E & R & T & k \\
\end{array} \]
```

**compare minimum in each subarray**

```
\[ aux[] = \begin{array}{cccccccc}
  & E & E & G & M & R & A & C & E & R & T & i & j \\
\end{array} \]
```
Merging demo

**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[]**

\[
\begin{array}{cccccccc}
A & C & E & E & \textcolor{red}{E} & A & C & E & R & T \\
\end{array}
\]

\(k\)

**compare minimum in each subarray**

**aux[]**

\[
\begin{array}{cccccccc}
E & E & \textcolor{red}{G} & M & R & A & C & E & R & T \\
\end{array}
\]

\(i\)

\(j\)
Merging demo

**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>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>
</tbody>
</table>
```

Merging demo

**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>j</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>i</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>j</td>
<td></td>
</tr>
</tbody>
</table>
Merging demo

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></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>k</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></td>
<td>i</td>
<td></td>
<td></td>
<td></td>
<td>j</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Goal. Given two sorted subarrays $a[\text{lo}]$ to $a[\text{mid}]$ and $a[\text{mid+1}]$ to $a[\text{hi}]$, replace with sorted subarray $a[\text{lo}]$ to $a[\text{hi}]$. 

**compare minimum in each subarray**

```plaintext
<table>
<thead>
<tr>
<th>a[]</th>
<th>A</th>
<th>C</th>
<th>E</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>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>
<td></td>
</tr>
</tbody>
</table>
```
**Merging demo**

**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  E  E  G  M  E  R  T
  - i

- **aux[]**
  - E  E  G  M  R  A  C  E  R  T
  - j

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

**compare minimum in each subarray**
Merging demo

**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>E</th>
<th>G</th>
<th>M</th>
<th>R</th>
<th>R</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<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>

One subarray exhausted, take from other

<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>j</td>
<td></td>
</tr>
</tbody>
</table>
Merging demo

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>k</td>
<td></td>
</tr>
</tbody>
</table>

one subarray exhausted, take from other

<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></td>
<td></td>
<td></td>
<td>i</td>
<td>j</td>
</tr>
</tbody>
</table>
Merging demo

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

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

one subarray exhausted, take from other

\[
\begin{array}{cccccccccc}
E & E & G & M & R & A & C & E & R & T \\
\end{array}
\]
Merging demo

*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>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>

One subarray exhausted, take from other.
**Merging demo**

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

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

**both subarrays exhausted, done**

\[
\begin{array}{cccccccc}
\text{aux[]} & E & E & G & M & R & A & C & E & R & T \\
\end{array}
\]
Merging demo

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

```
lo
```

```
hi
```

```
a[]
```

|   | A | C | E | E | E | G | M | R | R | T |
|---|---|---|---|---|---|---|---|---|---|

**sorted**
Merging demo

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

\[
\begin{array}{cccccccccc}
& & & & & & & & & \\
& & & lo & & & & hi \\
\begin{array}{cccccccccc}
A & C & E & E & E & G & M & R & R & T \\
& & & sorted & & & & \\
\end{array}
\end{array}
\]
Mergesort: Transylvanian-Saxon folk dance

http://www.youtube.com/watch?v=XaqR3G_NVoo
private static void merge(Comparable[] a, Comparable[] aux, int lo, int mid, int hi) {

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

    int i = lo, j = mid+1;
    for (int k = lo; k <= hi; k++)
    {
        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++];
    }
}
Assertions

**Assertion.** Statement to test assumptions about your program.
- Helps detect logic bugs.
- Documents code.

**Java assert statement.** Throws exception unless boolean condition is true.

```java
assert isSorted(a, lo, hi);
```

**Can enable or disable at runtime.** ⇒ No cost in production code.

```bash
% java -ea MyProgram   // enable assertions
% java -da MyProgram   // disable assertions (default)
```

**Best practices.** Use assertions to check internal invariants; assume assertions will be disabled in production code. ↩ do not use for external argument checking
Mergesort: Java implementation

```java
public class Merge {
    private static void merge(...)
    {
        /* 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)
    {
        Comparable[] aux = new Comparable[a.length];
        sort(a, aux, 0, a.length - 1);
    }
}
```

```
<table>
<thead>
<tr>
<th>lo</th>
<th>mid</th>
<th>hi</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>11</td>
<td>12</td>
</tr>
<tr>
<td>13</td>
<td>14</td>
<td>15</td>
</tr>
<tr>
<td>16</td>
<td>17</td>
<td>18</td>
</tr>
<tr>
<td>19</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
Mergesort: trace

merge(a, aux, 0, 0, 1)
merge(a, aux, 2, 2, 3)
merge(a, aux, 0, 1, 3)
merge(a, aux, 4, 4, 5)
merge(a, aux, 6, 6, 7)
merge(a, aux, 4, 5, 7)
merge(a, aux, 0, 3, 7)
merge(a, aux, 8, 8, 9)
merge(a, aux, 10, 10, 11)
merge(a, aux, 8, 9, 11)
merge(a, aux, 12, 12, 13)
merge(a, aux, 14, 14, 15)
merge(a, aux, 12, 13, 15)
merge(a, aux, 8, 11, 15)
merge(a, aux, 0, 7, 15)

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></th>
<th>insertion sort ($N^2$)</th>
<th>mergesort ($N \log N$)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>computer</td>
<td>thousand</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.
**Mergesort analysis: number of compares**

**Proposition.** Mergesort uses \( \leq N \lg N \) compares to sort an array of length \( N \).

**Pf sketch.** The number of compares \( C(N) \) to mergesort an array of length \( N \) satisfies the recurrence:

\[
C(N) \leq C([N/2]) + C([N/2]) + N - 1 \quad \text{for } N > 1, \text{ with } C(1) = 0.
\]

We solve this simpler recurrence, and assume \( N \) is a power of 2:

\[
D(N) = 2D(N/2) + N, \quad \text{for } N > 1, \text{ with } D(1) = 0.
\]

(result holds for all \( N \))

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

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

\[
\begin{align*}
T(N) &= N \lg N \\
N &= \text{top level} \\
2(N/2) &= \text{next level} \\
4(N/4) &= \text{next level} \\
8(N/8) &= \text{next level} \\
&\vdots
\end{align*}
\]
**Proposition.** If $D(N)$ satisfies $D(N) = 2D(N/2) + N$ for $N > 1$, with $D(1) = 0$, then $D(N) = N \lg N$.

**Pf by telescoping.** [assuming $N$ is a power of 2]

For $n > 1$:

\[
\frac{T(n)}{n} = \frac{2T(n/2)}{n} + 1
\]

\[
= \frac{T(n/2)}{n/2} + 1
\]

\[
= \frac{T(n/4)}{n/4} + 1 + 1
\]

\[\vdots\]

\[
= \frac{T(n/n)}{n/n} + \underbrace{1 + \cdots + 1}_{\log_2 n}
\]

\[
= \log_2 n
\]
Divide-and-conquer recurrence

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

**Pf by induction.** [assuming $N$ is a power of 2]

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

\[
D(2N) = 2D(N) + 2N
\]
given
\[
= 2N \lg N + 2N
\]
inductive hypothesis
\[
= 2N (\lg (2N) - 1) + 2N
\]
algebra
\[
= 2N \lg (2N)
\]
QED
Mergesort analysis: number of array accesses

**Proposition.** Mergesort uses \( \leq 6N \lg N \) array accesses to sort an array of length \( N \).

**Pf sketch.** The number of array accesses \( A(N) \) satisfies the recurrence:

\[
A(N) \leq A(\lfloor N/2 \rfloor) + A(\lfloor N/2 \rfloor) + 6N \quad \text{for } N > 1, \text{ with } A(1) = 0.
\]

**Key point.** Any algorithm with the following structure takes \( N \log N \) time:

```
public static void f(int N)
{
    if (N == 0) return;
    f(N/2);  // solve two problems
    f(N/2);  // of half the size
    linear(N);  // do a linear amount of work
}
```

**Notable algorithms.** FFT, hidden-line removal, Kendall-tau distance, ...
**Mergesort analysis: memory**

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

**Pf.** The array `aux[]` needs to be of length $N$ 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 1 (not hard).** Use `aux[]` array of length $\sim \frac{1}{2} N$ instead of $N$.

**Challenge 2 (very hard).** In-place merge. [Kronrod 1969].
Stability: mergesort

Proposition. Mergesort is stable.

```java
public class Merge
{
    private static void merge(...)
    {
        /* 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)
    {
        /* as before */
    }
}
```

Pf. Suffices to verify that merge operation is stable.
Stability: mergesort

Proposition. Merge operation is stable.

```java
private static void merge(...) {
    for (int k = lo; k <= hi; k++)
        aux[k] = a[k];

    int i = lo, j = mid+1;
    for (int k = lo; k <= hi; k++)
    {
        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++];
    }
}
```

<table>
<thead>
<tr>
<th>0 1 2 3 4</th>
<th>5 6 7 8 9 10</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1 A2 A3 B D</td>
<td>A4 A5 C E F G</td>
</tr>
</tbody>
</table>

Pf. Takes from left subarray if equal keys.
Mergesort: practical improvements

Use insertion sort for small subarrays.

- Mergesort has too much overhead for tiny subarrays.
- Cutoff to insertion sort for \( \approx 10 \) items.

```java
private static void sort(Comparable[] a, Comparable[] aux, int lo, int hi) {
    if (hi <= lo + CUTOFF - 1) {
        Insertion.sort(a, lo, hi);
        return;
    }

    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 largest item in first half $\leq$ smallest item in second half?
- Helps for partially-ordered arrays.

```
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(a, aux, lo, mid, hi);
}
```
Java 6 system sort

Basic algorithm for sorting objects = mergesort.

- Cutoff to insertion sort = 7.
- Stop-if-already-sorted test.
- Eliminate-the-copy-to-the-auxiliary-array trick.

Arrays.sort(a)

http://hg.openjdk.java.net/jdk6/jdk6/jdk/file/tip/src/share/classes/java/util/Arrays.java
Mergesort with cutoff to insertion sort: visualization
2.2 **Mergesort**

- mergesort
- *bottom-up mergesort*
- sorting complexity
- divide-and-conquer
Bottom-up mergesort

Basic plan.

- Pass through array, merging subarrays of size 1.
- Repeat for subarrays of size 2, 4, 8, ....
public class MergeBU
{
    private static void merge(…)
    { /* as before */ }

    public static void sort(Comparable[] a)
    {
        int N = a.length;
        Comparable[] 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, aux, lo, lo+sz-1, Math.min(lo+sz+sz-1, N-1));
    }
}
Mergesort: visualizations

- top-down mergesort (cutoff = 12)
- bottom-up mergesort (cutoff = 12)
Natural mergesort

Idea. Exploit pre-existing order by identifying naturally-occurring runs.

<table>
<thead>
<tr>
<th>input</th>
</tr>
</thead>
<tbody>
<tr>
<td>1  5  10  16  3  4  23  9  13  2  7  8  12  14</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>first run</th>
</tr>
</thead>
<tbody>
<tr>
<td>1  5  10  16  3  4  23  9  13  2  7  8  12  14</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>second run</th>
</tr>
</thead>
<tbody>
<tr>
<td>1  5  10  16  3  4  23  9  13  2  7  8  12  14</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>merge two runs</th>
</tr>
</thead>
<tbody>
<tr>
<td>1  3  4  5  10  16  23  9  13  2  7  8  12  14</td>
</tr>
</tbody>
</table>

Tradeoff. Fewer passes vs. extra compares per pass to identify runs.
Timsort

• Natural mergesort.
• Use binary insertion sort to make initial runs (if needed).
• A few more clever optimizations.

Intro
-----
This describes an adaptive, stable, natural mergesort, modestly called timsort (hey, I earned it <wink>). It has supernatural performance on many kinds of partially ordered arrays (less than \(\lg(N!)\) comparisons needed, and as few as \(N-1\)), yet as fast as Python's previous highly tuned samplesort hybrid on random arrays.

In a nutshell, the main routine marches over the array once, left to right, alternately identifying the next run, then merging it into the previous runs "intelligently". Everything else is complication for speed, and some hard-won measure of memory efficiency.
...

Consequence. Linear time on many arrays with pre-existing order.
Now widely used. Python, Java 7, GNU Octave, Android, ....

http://hg.openjdk.java.net/jdk7/jdk7/jdk/file/tip/src/share/classes/java/util/Arrays.java
Proving that Android’s, Java’s and Python’s sorting algorithm is broken (and showing how to fix it)

Tim Peters developed the Timsort hybrid sorting algorithm in 2002. It is a clever combination of ideas from merge sort and insertion sort, and designed to perform well on real world data. TimSort was first developed for Python, but later ported to Java (where it appears as java.util.Collections.sort and java.util.Arrays.sort) by Joshua Bloch (the designer of Java Collections who also pointed out that most binary search algorithms were broken). TimSort is today used as the default sorting algorithm for Android SDK, Sun’s JDK and OpenJDK. Given the popularity of these platforms this means that the number of computers, cloud services and mobile phones that use TimSort for sorting is well into the billions.

## Sorting summary

<table>
<thead>
<tr>
<th></th>
<th>inplace?</th>
<th>stable?</th>
<th>best</th>
<th>average</th>
<th>worst</th>
<th>remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>selection</td>
<td>✔</td>
<td></td>
<td>(\frac{1}{2} N^2)</td>
<td>(\frac{1}{2} N^2)</td>
<td>(\frac{1}{2} N^2)</td>
<td>(N) exchanges</td>
</tr>
<tr>
<td>insertion</td>
<td>✔</td>
<td>✔</td>
<td>(N)</td>
<td>(\frac{1}{4} N^2)</td>
<td>(\frac{1}{2} N^2)</td>
<td>use for small (N) or partially ordered</td>
</tr>
<tr>
<td>shell</td>
<td>✔</td>
<td></td>
<td>(N \log_3 N)</td>
<td>?</td>
<td>(c N^{3/2})</td>
<td>tight code; subquadratic</td>
</tr>
<tr>
<td>merge</td>
<td>✔</td>
<td></td>
<td>(\frac{1}{2} N \log N)</td>
<td>(N \log N)</td>
<td>(N \log N)</td>
<td>(N \log N) guarantee; stable</td>
</tr>
<tr>
<td>timsort</td>
<td>✔</td>
<td>✔</td>
<td>(N)</td>
<td>(N \log N)</td>
<td>(N \log N)</td>
<td>improves mergesort when preexisting order</td>
</tr>
<tr>
<td>?</td>
<td>✔</td>
<td>✔</td>
<td>(N)</td>
<td>(N \log N)</td>
<td>(N \log N)</td>
<td>holy sorting grail</td>
</tr>
</tbody>
</table>
2.2 MERGESORT

- mergesort
- bottom-up mergesort
- sorting complexity
- divide-and-conquer
**Complexity of sorting**

**Computational complexity.** Framework to study efficiency of algorithms for solving a particular problem $X$.

**Model of computation.** Allowable operations.

**Cost model.** Operation counts.

**Upper bound.** Cost guarantee provided by *some* algorithm for $X$.

**Lower bound.** Proven limit on cost guarantee of *all* algorithms for $X$.

**Optimal algorithm.** Algorithm with best possible cost guarantee for $X$.

<table>
<thead>
<tr>
<th>Model of Computation</th>
<th>Decision Tree</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cost Model</td>
<td># Comparues</td>
</tr>
<tr>
<td>Upper Bound</td>
<td>$\sim N \lg N$ from mergesort</td>
</tr>
<tr>
<td>Lower Bound</td>
<td>?</td>
</tr>
<tr>
<td>Optimal Algorithm</td>
<td>?</td>
</tr>
</tbody>
</table>

Can access information only through compares (e.g., Java Comparable framework)

lower bound ~ upper bound
Decision tree (for 3 distinct keys a, b, and c)

- **a < b**
  - yes: `b < c`
  - no: `a < c`

- **b < c**
  - yes: `a < c`
  - no: `

- **a < c**
  - yes: `b < c`
  - no: `

Height of tree = worst-case number of compares

each leaf corresponds to one (and only one) ordering; (at least) one leaf for each possible ordering
Proposition. Any compare-based sorting algorithm must use at least \( \lg (N!) \sim N \lg N \) compares in the worst-case.

\[ \lg (N!) \sim N \lg N \]

\textbf{Pf.}

- Assume array consists of \( N \) distinct values \( a_1 \) through \( a_N \).
- Worst case dictated by \textbf{height} \( h \) of decision tree.
- Binary tree of height \( h \) has at most \( 2^h \) leaves.
- \( N! \) different orderings \( \Rightarrow \) at least \( N! \) leaves.

\[ \text{at least } N! \text{ leaves} \]

\[ \text{no more than } 2^h \text{ leaves} \]
Compare-based lower bound for sorting

**Proposition.** Any compare-based sorting algorithm must use at least
\[ \lg (N!) \sim N \lg N \] compares in the worst-case.

**Pf.**
- Assume array consists of \( N \) distinct values \( a_1 \) through \( a_N \).
- Worst case dictated by height \( h \) of decision tree.
- Binary tree of height \( h \) has at most \( 2^h \) leaves.
- \( N! \) different orderings \( \Rightarrow \) at least \( N! \) leaves.

\[
2^h \geq \# \text{ leaves} \geq N! \\
\Rightarrow h \geq \lg (N!) \sim N \lg N
\]
Complexity of sorting

Model of computation. Allowable operations.
Cost model. Operation count(s).
Upper bound. Cost guarantee provided by some algorithm for \( X \).
Lower bound. Proven limit on cost guarantee of all algorithms for \( X \).
Optimal algorithm. Algorithm with best possible cost guarantee for \( X \).

<table>
<thead>
<tr>
<th>model of computation</th>
<th>decision tree</th>
</tr>
</thead>
<tbody>
<tr>
<td>cost model</td>
<td># compares</td>
</tr>
<tr>
<td>upper bound</td>
<td>( \sim N \lg N )</td>
</tr>
<tr>
<td>lower bound</td>
<td>( \sim N \lg N )</td>
</tr>
<tr>
<td>optimal algorithm</td>
<td>mergesort</td>
</tr>
</tbody>
</table>

First goal of algorithm design: optimal algorithms.
Complexity results in context

**Compared?** Mergesort is optimal with respect to number compares.
**Space?** Mergesort is not optimal with respect to space usage.

**Lessons.** Use theory as a guide.

**Ex.** Design sorting algorithm that guarantees $\sim \frac{1}{2} N \lg N$ compares?
**Ex.** Design sorting algorithm that is both time- and space-optimal?
Lower bound may not hold if the algorithm can take advantage of:

- The initial order of the input.
  Ex: insertion sort requires only a linear number of compares on partially-sorted arrays.

- The distribution of key values.
  Ex: 3-way quicksort requires only a linear number of compares on arrays with a constant number of distinct keys. [stay tuned]

- The representation of the keys.
  Ex: radix sorts require no key compares — they access the data via character/digit compares.
Commonly-used notations in the theory of algorithms

<table>
<thead>
<tr>
<th>notation</th>
<th>provides</th>
<th>example</th>
<th>shorthand for</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tilde</td>
<td>leading term</td>
<td>$\sim \frac{1}{2}N^2$</td>
<td>$\frac{1}{2}N^2$ $\frac{1}{2}N^2 + 22N \log N + 3N$</td>
</tr>
<tr>
<td>Big Theta</td>
<td>order of growth</td>
<td>$\Theta(N^2)$</td>
<td>$\frac{1}{2}N^2$ $10N^2$ $5N^2 + 22N \log N + 3N$</td>
</tr>
<tr>
<td>Big O</td>
<td>upper bound</td>
<td>$O(N^2)$</td>
<td>$10N^2$ $100N$ $22N \log N + 3N$</td>
</tr>
<tr>
<td>Big Omega</td>
<td>lower bound</td>
<td>$\Omega(N^2)$</td>
<td>$\frac{1}{2}N^2$ $N^5$ $N^3 + 22N \log N + 3N$</td>
</tr>
</tbody>
</table>