I have changed the homework assignments to be due on Thursdays rather than Tuesdays, to make better use of office hours. All HWs are still due before class at 10:30am.

HW3 will be released today. Reminder: DO NOT SHARE CODE OR DISCUSS YOUR IMPLEMENTATION. Do not publish or look online.

Grades for HW1 have been returned. Talk to your recitation TAs if you have questions about the rubric or about the re-grading policy.

Be sure to do all of the readings. They are listed on the Lectures page of the CIS 121 web site. The homework and the exams assume that you have done the assigned readings in addition to attending the lectures.

I will be traveling on Tuesday. The TAs have volunteered to teach a course focusing on reviewing HW1 and HW2, and on Big O and recurrences.
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

**Mergesort example**

<table>
<thead>
<tr>
<th>input</th>
<th>MERGESORT EXAMPLE</th>
</tr>
</thead>
<tbody>
<tr>
<td>sort left half</td>
<td>E E G M O R R S</td>
</tr>
<tr>
<td>sort right half</td>
<td>E E G M O R R S</td>
</tr>
<tr>
<td>merge results</td>
<td>A E E E E G L M M</td>
</tr>
<tr>
<td></td>
<td>O P R R S T X</td>
</tr>
</tbody>
</table>
Abstract in-place merge 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 showing in-place merge](image-url)

- $a[]$: E E G M R | A C E R T
- Sorted: $a[lo]$ to $a[mid]$ and $a[mid+1]$ to $a[hi]$
**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></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>
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 \( a[hi] \).
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 \( a[hi] \).

### compare minimum in each subarray

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

  \( k \)

- **aux[]**
  - \( E \) \( E \) \( G \) \( M \) \( R \) \( A \) \( C \) \( E \) \( R \) \( T \)

  \( i \) \( j \)
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 $a[hi]$.

---

**compare minimum in each subarray**

<table>
<thead>
<tr>
<th>a[]</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>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></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></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>
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>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>
**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 of merging process]

**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>
</table>
```

`i` | `j`
**Merging demo**

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

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

\[
\begin{array}{cccccccccccc}
\text{k} \\
\end{array}
\]

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

**compare minimum in each subarray**

\[
\begin{array}{cccccccccccc}
i \\
\end{array}
\]

\[
\begin{array}{cccccccccccc}
j \\
\end{array}
\]
**Goal.** Given two sorted subarrays $a[lo]$ to $a[mid]$ and $a[mid+1]$ to $a[hi]$, replace with sorted subarray $a[lo]$ to $a[hi]$.

**compare minimum in each subarray**

![Diagram showing merging process]
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 $a[hi]$.

**compare minimum in each subarray**

<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></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>

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

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 \( a[hi] \).

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

**compare minimum in each subarray**

\[
\begin{array}{cccccccc}
E & E & G & M & R & A & C & E & R & T \\
i & j
\end{array}
\]
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 $a[hi]$.

<table>
<thead>
<tr>
<th>a[]</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
</tr>
<tr>
<td>C</td>
</tr>
<tr>
<td>E</td>
</tr>
<tr>
<td>E</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>

$k$

compare minimum in each subarray

<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>

$i$ $j$
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></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>j</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>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>
**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></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>
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 \(a[hi]\).

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

\(k\)

**compare minimum in each subarray**

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

\(i\)

\(j\)
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>G</th>
<th>M</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>(j)</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></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>k</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>

**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>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>
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>R</td>
<td>R</td>
<td>T</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></td>
<td></td>
<td></td>
<td>j</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>E</th>
<th>G</th>
<th>M</th>
<th>R</th>
<th>R</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a[i]$</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>$a[hi]$</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>

**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>$aux[i]$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$aux[j]$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

$k$
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 $a[hi]$.

**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 $a[hi]$.

<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>
</table>
| k

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>
</table>
| 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].`

---

**one subarray exhausted, take from other**

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

`a[]`:
- A
- C
- E
- E
- E
- E
- G
- M
- R
- R
- T

`aux[]`:
- E
- E
- G
- M
- R
- A
- C
- E
- R
- T

k

i

j
Merging demo

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

Both subarrays exhausted, done

Both subarrays exhausted, done
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 of merging two sorted subarrays](image-url)
Abstract in-place merge 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 showing two sorted subarrays](image-url)
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]; // copy

    int i = lo, j = mid+1;
    for (int k = lo; k <= hi; k++)
    {
        if       (i > mid) a[k] = aux[j++]; // merge
        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(...) {
        /* 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, 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);
    }
}
```
Mergesort: trace

<table>
<thead>
<tr>
<th>a[]</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>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>E</td>
<td>M</td>
<td>R</td>
<td>G</td>
<td>E</td>
<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>E</td>
</tr>
<tr>
<td></td>
<td>E</td>
<td>M</td>
<td>G</td>
<td>R</td>
<td>E</td>
<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>E</td>
</tr>
<tr>
<td></td>
<td>E</td>
<td>G</td>
<td>M</td>
<td>R</td>
<td>E</td>
<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>E</td>
</tr>
<tr>
<td></td>
<td>E</td>
<td>G</td>
<td>M</td>
<td>R</td>
<td>E</td>
<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>E</td>
</tr>
<tr>
<td></td>
<td>E</td>
<td>G</td>
<td>M</td>
<td>R</td>
<td>E</td>
<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>E</td>
</tr>
<tr>
<td></td>
<td>E</td>
<td>G</td>
<td>M</td>
<td>R</td>
<td>E</td>
<td>O</td>
<td>R</td>
<td>S</td>
<td>T</td>
<td>E</td>
<td>X</td>
<td>A</td>
<td>M</td>
<td>P</td>
<td>L</td>
<td>E</td>
</tr>
<tr>
<td></td>
<td>E</td>
<td>E</td>
<td>G</td>
<td>M</td>
<td>O</td>
<td>R</td>
<td>R</td>
<td>S</td>
<td>T</td>
<td>E</td>
<td>X</td>
<td>A</td>
<td>M</td>
<td>P</td>
<td>L</td>
<td>E</td>
</tr>
<tr>
<td></td>
<td>E</td>
<td>E</td>
<td>G</td>
<td>M</td>
<td>O</td>
<td>R</td>
<td>R</td>
<td>S</td>
<td>E</td>
<td>T</td>
<td>X</td>
<td>A</td>
<td>M</td>
<td>P</td>
<td>L</td>
<td>E</td>
</tr>
<tr>
<td></td>
<td>E</td>
<td>E</td>
<td>G</td>
<td>M</td>
<td>O</td>
<td>R</td>
<td>R</td>
<td>S</td>
<td>E</td>
<td>T</td>
<td>A</td>
<td>X</td>
<td>M</td>
<td>P</td>
<td>L</td>
<td>E</td>
</tr>
<tr>
<td></td>
<td>E</td>
<td>E</td>
<td>G</td>
<td>M</td>
<td>O</td>
<td>R</td>
<td>R</td>
<td>S</td>
<td>A</td>
<td>E</td>
<td>T</td>
<td>X</td>
<td>M</td>
<td>P</td>
<td>L</td>
<td>E</td>
</tr>
<tr>
<td></td>
<td>E</td>
<td>E</td>
<td>G</td>
<td>M</td>
<td>O</td>
<td>R</td>
<td>R</td>
<td>S</td>
<td>A</td>
<td>E</td>
<td>T</td>
<td>X</td>
<td>M</td>
<td>P</td>
<td>E</td>
<td>L</td>
</tr>
<tr>
<td></td>
<td>E</td>
<td>E</td>
<td>G</td>
<td>M</td>
<td>O</td>
<td>R</td>
<td>R</td>
<td>S</td>
<td>A</td>
<td>E</td>
<td>T</td>
<td>X</td>
<td>E</td>
<td>L</td>
<td>M</td>
<td>P</td>
</tr>
<tr>
<td></td>
<td>E</td>
<td>E</td>
<td>G</td>
<td>M</td>
<td>O</td>
<td>R</td>
<td>R</td>
<td>S</td>
<td>A</td>
<td>E</td>
<td>E</td>
<td>L</td>
<td>M</td>
<td>P</td>
<td>T</td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>A</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>G</td>
<td>L</td>
<td>M</td>
<td>M</td>
<td>O</td>
<td>P</td>
<td>R</td>
<td>R</td>
<td>S</td>
<td>T</td>
<td>X</td>
</tr>
</tbody>
</table>

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
Which of the following subarray lengths will occur when running mergesort on an array of length 12?

A. \{ 1, 2, 3, 4, 6, 8, 12 \}

B. \{ 1, 2, 3, 6, 12 \}

C. \{ 1, 2, 4, 8, 12 \}

D. \{ 1, 3, 6, 9, 12 \}

E. I don't know.
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>thousand</th>
<th>million</th>
<th>billion</th>
<th>thousand</th>
<th>million</th>
<th>billion</th>
</tr>
</thead>
<tbody>
<tr>
<td>home</td>
<td>instant</td>
<td>2.8 hours</td>
<td>317 years</td>
<td>instant</td>
<td>1 second</td>
<td>18 min</td>
</tr>
<tr>
<td>super</td>
<td>instant</td>
<td>1 second</td>
<td>1 week</td>
<td>instant</td>
<td>instant</td>
<td>instant</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:

\[
D(N) = 2D(N/2) + N, \text{ for } N > 1, \text{ with } D(1) = 0.
\]
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 picture.** [assuming \( N \) is a power of 2]

\[
T(N) = N \lg N
\]
Mergesort analysis: number of array accesses

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

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

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

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

```java
public static void f(int N)
{
    if (N == 0) return;
    f(N/2); \hspace{1cm} \text{solve two problems}
    f(N/2); \hspace{1cm} \text{of half the size}
    linear(N); \hspace{1cm} \text{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 \( \text{aux}[] \) 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 \( \text{aux}[] \) array of length \( \sim \frac{1}{2} N \) instead of \( N \).

**Challenge 2 (very hard).** In-place merge. [Kronrod 1969].
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</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A₁</td>
<td>A₂</td>
<td>A₃</td>
<td>B</td>
<td>D</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>A₄</td>
<td>A₅</td>
<td>C</td>
<td>E</td>
<td>F</td>
<td>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 ≤ 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, ....

<table>
<thead>
<tr>
<th>sz</th>
<th>merge(a, aux, 0, 0, 1)</th>
<th>merge(a, aux, 2, 2, 3)</th>
<th>merge(a, aux, 4, 4, 5)</th>
<th>merge(a, aux, 6, 6, 7)</th>
<th>merge(a, aux, 10, 10, 11)</th>
<th>merge(a, aux, 12, 12, 13)</th>
<th>merge(a, aux, 14, 14, 15)</th>
</tr>
</thead>
</table>

sz = 1

sz = 2

sz = 4

sz = 8
Bottom-up mergesort: Java implementation

```java
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));
  }
}
```

**Bottom line.** Simple and non-recursive version of mergesort.
Mergesort: visualizations

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

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

**input**

1  5  10  16  3  4  23  9  13  2  7  8  12  14

**first run**

1  5  10  16  3  4  23  9  13  2  7  8  12  14

**second run**

1  5  10  16  3  4  23  9  13  2  7  8  12  14

**merge two runs**

1  3  4  5  10  16  23  9  13  2  7  8  12  14

**Tradeoff.** Fewer passes vs. extra compares per pass to identify runs.
**Tim sort**

- 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 *tim sort* (hey, I earned it <wink>). It has supernatural performance on many kinds of partially ordered arrays (less than \(\log(N)\) 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, ...
Timsort bug

Envisage: Engineering Virtualized Services

Proving that Android’s, Java’s and Python’s sorting algorithm is broken (and showing how to fix it)

© February 24, 2015  Envisage  Written by Stijn de Gouw. $s

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>$\frac{1}{2} N \lg N$</td>
<td>$N \lg N$</td>
<td>$N \lg N$</td>
<td>$N \log N$ guarantee; stable</td>
</tr>
<tr>
<td>timsort</td>
<td>✔️</td>
<td>✔️</td>
<td>$N$</td>
<td>$N \lg N$</td>
<td>$N \lg N$</td>
<td>improves mergesort when preexisting order</td>
</tr>
<tr>
<td></td>
<td>✔️</td>
<td>✔️</td>
<td>$N$</td>
<td>$N \lg N$</td>
<td>$N \lg N$</td>
<td>holy sorting grail</td>
</tr>
</tbody>
</table>
2.2 Mergesort

- mergesort
- bottom-up mergesort
- sorting complexity
- divide-and-conquer
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 $X$. lower bound ~ upper bound

<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>$\sim N \lg N$ from mergesort</td>
</tr>
<tr>
<td>lower bound</td>
<td>?</td>
</tr>
</tbody>
</table>
| optimal algorithm    | ?             | can access information only through compares (e.g., Java Comparable framework)
Decision tree (for 3 distinct keys a, b, and c)

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

**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.

\begin{align*}
\text{Pf.} & \\
\bullet & \text{Assume array consists of } N \text{ distinct values } a_1 \text{ through } a_N. \\
\bullet & \text{Worst case dictated by height } h \text{ of decision tree.} \\
\bullet & \text{Binary tree of height } h \text{ has at most } 2^h \text{ leaves.} \\
\bullet & N! \text{ different orderings } \Rightarrow \text{ at least } N! \text{ leaves.}
\end{align*}
**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

**Comparisons?** 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>
</table>
| **Tilde** | leading term                  | $\sim \frac{1}{2} N^2$    | $\frac{1}{2} N^2$  
                                        |                               | $\frac{1}{2} N^2 + 22 N \log N + 3 N$ |
| **Big Theta** | order of growth             | $\Theta(N^2)$               | $\frac{1}{2} N^2$  
                                        |                               | $10 N^2$  
                                        | $5 N^2 + 22 N \log N + 3 N$ |
| **Big O**  | upper bound                   | $O(N^2)$                    | $10 N^2$  
                                        |                               | $100 N$  
                                        | $22 N \log N + 3 N$ |
| **Big Omega** | lower bound                  | $\Omega(N^2)$             | $\frac{1}{2} N^2$  
                                        |                               | $N^5$  
                                        | $N^3 + 22 N \log N + 3 N$ |
1.4 Analysis of Algorithms

- introduction
- observations
- mathematical models
- order-of-growth classifications
- theory of algorithms
- memory
Types of analyses

**Best case.** Lower bound on cost.
- Determined by “easiest” input.
- Provides a goal for all inputs.

**Worst case.** Upper bound on cost.
- Determined by “most difficult” input.
- Provides a guarantee for all inputs.

**Average case.** Expected cost for random input.
- Need a model for “random” input.
- Provides a way to predict performance.

---

**Ex 1.** Array accesses for brute-force 3-Sum.
- Best: \( \sim \frac{1}{2} N^3 \)
- Average: \( \sim \frac{1}{2} N^3 \)
- Worst: \( \sim \frac{1}{2} N^3 \)

**Ex 2.** Compares for binary search.
- Best: \( \sim 1 \)
- Average: \( \sim \log N \)
- Worst: \( \sim \log N \)
Types of analyses

**Best case.** Lower bound on cost.

**Worst case.** Upper bound on cost.

**Average case.** “Expected” cost.

Actual data might not match input model?

- Need to understand input to effectively process it.
- Approach 1: design for the worst case.
- Approach 2: randomize, depend on probabilistic guarantee.
Theory of algorithms

Goals.

• Establish “difficulty” of a problem.
• Develop “optimal” algorithms.

Approach.

• Suppress details in analysis: analyze “to within a constant factor.”
• Eliminate variability in input model: focus on the worst case.

Upper bound. Performance guarantee of algorithm for any input.
Lower bound. Proof that no algorithm can do better.
Optimal algorithm. Lower bound = upper bound (to within a constant factor).
### Commonly-used notations in the theory of algorithms

<table>
<thead>
<tr>
<th>notation</th>
<th>provides</th>
<th>example</th>
<th>shorthand for</th>
<th>used to</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Big Theta</strong></td>
<td>asymptotic order of growth</td>
<td>$\Theta(N^2)$</td>
<td>$\frac{1}{2} N^2$ $10 N^2$ $5 N^2 + 22 N \log N + 3N$ $\vdots$</td>
<td>classify algorithms</td>
</tr>
<tr>
<td><strong>Big O</strong></td>
<td>$\Theta(N^2)$ and smaller</td>
<td>$O(N^2)$</td>
<td>$10 N^2$ $100 N$ $22 N \log N + 3N$ $\vdots$</td>
<td>develop upper bounds</td>
</tr>
<tr>
<td><strong>Big Omega</strong></td>
<td>$\Theta(N^2)$ and larger</td>
<td>$\Omega(N^2)$</td>
<td>$\frac{1}{2} N^2$ $N^5$ $N^3 + 22 N \log N + 3N$ $\vdots$</td>
<td>develop lower bounds</td>
</tr>
</tbody>
</table>
Goals.
- Establish “difficulty” of a problem and develop “optimal” algorithms.
- Ex. 1-SUM = “Is there a 0 in the array?”

Upper bound. A specific algorithm.
- Running time of the optimal algorithm for 1-SUM is $O(N)$.

Lower bound. Proof that no algorithm can do better.
- Ex. Have to examine all $N$ entries (any unexamined one might be 0).
- Running time of the optimal algorithm for 1-SUM is $\Omega(N)$.

Optimal algorithm.
- Lower bound equals upper bound (to within a constant factor).
- Ex. Brute-force algorithm for 1-SUM is optimal: its running time is $\Theta(N)$.
Theory of algorithms: example 2

Goals.
- Establish “difficulty” of a problem and develop “optimal” algorithms.
- Ex. 3-Sum.

Upper bound. A specific algorithm.
- Ex. Brute-force algorithm for 3-Sum.
- Running time of the optimal algorithm for 3-Sum is $O(N^3)$. 
Goals.
- Establish “difficulty” of a problem and develop “optimal” algorithms.
- Ex. 3-SUM.

Upper bound. A specific algorithm.
- Ex. Improved algorithm for 3-SUM.
- Running time of the optimal algorithm for 3-SUM is $O(N^2 \log N)$.

Lower bound. Proof that no algorithm can do better.
- Ex. Have to examine all $N$ entries to solve 3-SUM.
- Running time of the optimal algorithm for solving 3-SUM is $\Omega(N)$.

Open problems.
- Optimal algorithm for 3-SUM?
- Subquadratic algorithm for 3-SUM?
- Quadratic lower bound for 3-SUM?
Algorithm design approach

Start.
• Develop an algorithm.
• Prove a lower bound.

Gap?
• Lower the upper bound (discover a new algorithm).
• Raise the lower bound (more difficult).

Golden Age of Algorithm Design.
• 1970s–.
  • Steadily decreasing upper bounds for many important problems.
  • Many known optimal algorithms.

Caveats.
• Overly pessimistic to focus on worst case?
• Need better than “to within a constant factor” to predict performance.
Commonly-used notations in the theory of algorithms

<table>
<thead>
<tr>
<th>notation</th>
<th>provides</th>
<th>example</th>
<th>shorthand for</th>
<th>used to</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tilde</td>
<td>leading term</td>
<td>$\sim 10 , N^2$</td>
<td>$10 , N^2$ $10 , N^2 + 22 , N \log N$ $10 , N^2 + 2 , N + 37$</td>
<td>provide approximate model</td>
</tr>
<tr>
<td>Big Theta</td>
<td>asymptotic order of growth</td>
<td>$\Theta(N^2)$</td>
<td>$\frac{1}{2} , N^2$ $10 , N^2$ $5 , N^2 + 22 , N \log N + 3N$</td>
<td>classify algorithms</td>
</tr>
<tr>
<td>Big Oh</td>
<td>$\Theta(N^2)$ and smaller</td>
<td>$O(N^2)$</td>
<td>$10 , N^2$ $100 , N$ $22 , N \log N + 3N$</td>
<td>develop upper bounds</td>
</tr>
<tr>
<td>Big Omega</td>
<td>$\Theta(N^2)$ and larger</td>
<td>$\Omega(N^2)$</td>
<td>$\frac{1}{2} , N^2$ $N^5$ $N^3 + 22 , N \log N + 3N$</td>
<td>develop lower bounds</td>
</tr>
</tbody>
</table>

**Common mistake.** Interpreting big-Oh as an approximate model.

**This course.** Focus on approximate models: use Tilde-notation