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. [this lecture]

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 the two halves.

<table>
<thead>
<tr>
<th>input</th>
<th>MERGESORT EXAMPLE</th>
</tr>
</thead>
<tbody>
<tr>
<td>sort left half</td>
<td>EEGMORS</td>
</tr>
<tr>
<td>sort right half</td>
<td>EEGMORS</td>
</tr>
<tr>
<td>merge results</td>
<td>AEEEGLMMPRTX</td>
</tr>
</tbody>
</table>

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

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

sorted

sorted
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++];
        }
}

Bad programming practice.
Used here to save vertical space.
Equivalent to:
{
    a[k] = aux[j];
    j++;
}
Mergesort quiz 1

How many calls does \texttt{merge()} make to \texttt{less()} in order to merge two sorted subarrays, each of length $n/2$, into a sorted array of length $n$?

A. $\sim \frac{1}{4} n$ to $\sim \frac{1}{2} n$

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

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

D. $\sim n$
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)

a[]

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
MERGESORTEXAMPLE
MERGESORTEXAMPLE
MERGESORTEXAMPLE
MERGESORTEXAMPLE
MERGESORTEXAMPLE
MERGESORTEXAMPLE
MERGESORTEXAMPLE
MERGESORTEXAMPLE
MERGESORTEXAMPLE
MERGESORTEXAMPLE
MERGESORTEXAMPLE
MERGESORTEXAMPLE
MERGESORTEXAMPLE
MERGESORTEXAMPLE
MERGESORTEXAMPLE

lo hi

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 }
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><strong>computer</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>home</strong></td>
<td>instant</td>
<td>2.8 hours</td>
</tr>
<tr>
<td><strong>super</strong></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 any array of length $n$.

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

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

We solve this simpler recurrence, and assume $n$ is a power of 2:

$$D(n) = 2D(n/2) + n, \text{ for } n > 1, \text{ with } D(1) = 0.$$  

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 $D(n) = n \log n$.

**Pf by picture.** [assuming $n$ is a power of 2]
Key point. Any algorithm with the following structure takes $\Theta(n \log n)$ time:

```java
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 examples. FFT, hidden-line removal, Kendall-tau distance, ...
Mergesort analysis: number of array accesses

Proposition. Mergesort uses $\leq 6n \lg n$ array accesses to sort any array of length $n$.

Pf sketch. The number of array accesses $A(n)$ satisfies the recurrence:

$$A(n) \leq A([n/2]) + A([n/2]) + 6n \text{ for } n > 1, \text{ with } A(1) = 0.$$  

[Rest of the proof is similar to the analysis of number of compares.]
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.

<table>
<thead>
<tr>
<th>two sorted subarrays</th>
</tr>
</thead>
<tbody>
<tr>
<td>A C D G H I M N U V</td>
</tr>
<tr>
<td>B E F J O P Q R S T</td>
</tr>
</tbody>
</table>

merged result

| A B C D E F G H I J M N O P Q R S T U V |

“Essentially no extra memory”

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

**Ex.** Insertion sort and selection sort.

**Challenge 1 (not hard).** Use `aux[]` array of length $\sim \frac{1}{2} n$ instead of $n$.

**Challenge 2 (very hard).** In-place merge.
Is our implementation of mergesort stable?

A. Yes.
B. No, but it can be easily modified to be stable.
C. No, mergesort is inherently unstable.
D. *I don’t remember what stability means.*

A sorting algorithm is stable if it preserves the relative order of equal keys.

---

**Input:** C A₁ B A₂ A₃

**Sorted:** A₃ A₁ A₂ B C

*not stable*
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></th>
<th>A1</th>
<th>A2</th>
<th>A3</th>
<th>B</th>
<th>D</th>
<th></th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</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>1</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>2</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>3</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>4</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>5</td>
<td>A4</td>
<td>A5</td>
<td>C</td>
<td>E</td>
<td>F</td>
<td>G</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></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.

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

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 6: Arrays.sort() uses mergesort for sorting objects, with the above tricks.
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 = 1</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, 8, 8, 9)</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>

<table>
<thead>
<tr>
<th>sz = 2</th>
<th>merge(a, aux, 0, 1, 3)</th>
<th>merge(a, aux, 4, 5, 7)</th>
<th>merge(a, aux, 8, 9, 11)</th>
<th>merge(a, aux, 12, 13, 15)</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>sz = 4</th>
<th>merge(a, aux, 0, 3, 7)</th>
<th>merge(a, aux, 8, 11, 15)</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>sz = 8</th>
<th>merge(a, aux, 0, 7, 15)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A E E E E G L M M O P R R S T X</td>
</tr>
</tbody>
</table>
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)
Which is faster in practice for \( n = 2^{20} \), top-down mergesort or bottom-up mergesort?

A. Top-down (recursive) mergesort.

B. Bottom-up (non-recursive) mergesort.

C. No observable difference.

D. I don't know.

Hint: the answer depends on concepts you’ll learn in COS 217.
## 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>?</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*
## 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>comparison 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>
<tr>
<td>optimal algorithm</td>
<td>?</td>
</tr>
</tbody>
</table>

lower bound \( \sim \) upper bound

can access information only through compares (e.g., Java Comparable framework)
Comparison tree (for 3 distinct keys a, b, and c)

- a < b
- b < c
- a < c

Each reachable leaf corresponds to one (and only one) ordering; exactly one reachable leaf for each possible ordering.

Height of pruned comparison tree = worst-case number of compares.
Compare-based lower bound for sorting

Proposition. Any compare-based sorting algorithm must make 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 number of compares = height $h$ of pruned comparison tree.
- Binary tree of height $h$ has $\leq 2^h$ leaves.
- $n!$ different orderings $\Rightarrow$ $n!$ reachable leaves.
Compare-based lower bound for sorting

**Proposition.** Any compare-based sorting algorithm must make 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 number of compares = height \( h \) of pruned comparison tree.
- Binary tree of height \( h \) has \( \leq 2^h \) leaves.
- \( n! \) different orderings \( \Rightarrow n! \) reachable leaves.

\[
2^h \geq \# \text{ reachable leaves} = n! \\
\Rightarrow h \geq \lg(n!) \\
\sim n \lg n
\]

\[\text{Stirling’s formula}\]
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>comparison 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

**Compares?** 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?
Q. Why doesn’t this Skittles sorter violate the sorting lower bound?

https://www.youtube.com/watch?v=tSEHDBSvnVo
Lower bound may not hold if the algorithm can take advantage of:

- The initial order of the input array.
  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. [stay tuned]
A brief history of sorting: Hollerith census tabulator (1890s)
IBM card sorter (1940s)
### Big O notation (and cousins)

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

What’s wrong with this statement? How would you correct it?
Any compare-based sorting algorithm must make at least $O(n \log n)$ compares in the worst case.

“At least $O(n \log n)$” is a nonsensical (but frequently heard) expression.

Correct: any compare-based sorting algorithm must make $\Omega(n \log n)$ compares in the worst case.

No need to say “at least $\Omega(n \log n)$” — that’s implicit in the definition of $\Omega$. 
2.2 Mergesort

- mergesort
- bottom-up mergesort
- sorting complexity
- divide-and-conquer
Problem. Given a singly linked list, rearrange its nodes in sorter order.

Version 1. Linearithmic time, linear extra space.
Version 2. Linearithmic time, logarithmic (or constant) extra space.