2.2 Mergesort

- mergesort
- bottom-up mergesort
- sorting complexity
- comparators
Two classic sorting algorithms

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.
• Java sort for objects.
• Perl, Python stable sort.

Quicksort.
• Java sort for primitive types.
• C qsort, Unix, g++, Visual C++, Python.
- mergesort
- bottom-up mergesort
- sorting complexity
- comparators
Mergesort

Basic plan.

• Divide array into two halves.
• Recursively sort each half.
• Merge two halves.
Merging

Q. How to combine two sorted subarrays into a sorted whole.
A. Use an auxiliary array.

Abstract in-place merge trace
Merging: Java implementation

```java
private static void merge(Comparable[] a, int lo, int mid, int hi)
{
    assert isSorted(a, lo, mid);   // precondition: a[lo..mid] sorted
    assert isSorted(a, mid+1, hi); // precondition: a[mid+1..hi] sorted

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

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

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

```text
<table>
<thead>
<tr>
<th>aux[]</th>
<th>lo</th>
<th>i</th>
<th>mid</th>
<th>j</th>
<th>hi</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A</td>
<td>G</td>
<td>L</td>
<td>O</td>
<td>R</td>
</tr>
<tr>
<td></td>
<td>H</td>
<td>I</td>
<td>M</td>
<td>S</td>
<td>T</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>a[]</th>
<th>lo</th>
<th>i</th>
<th>mid</th>
<th>j</th>
<th>hi</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A</td>
<td>G</td>
<td>H</td>
<td>I</td>
<td>L</td>
</tr>
<tr>
<td></td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
</tr>
</tbody>
</table>
```
Assertions

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

**Java assert statement.** Throws an 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 to check internal invariants. Assume assertions will be disabled in production code (e.g., don't use for external argument-checking).
public class Merge
{
   private static Comparable[] aux;

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

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

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

Trace of merge results for top-down mergesort

result after recursive call
Mergesort animation

50 random elements

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

50 reverse-sorted elements

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

Running time estimates:
- Home pc executes $10^8$ comparisons/second.
- Supercomputer executes $10^{12}$ comparisons/second.

<table>
<thead>
<tr>
<th>Computer</th>
<th>Insertion Sort ($N^2$)</th>
<th>Mergesort ($N \log N$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>home</td>
<td>instant</td>
<td>instant</td>
</tr>
<tr>
<td></td>
<td>2.8 hours</td>
<td>1 second</td>
</tr>
<tr>
<td></td>
<td>317 years</td>
<td>18 min</td>
</tr>
<tr>
<td>super</td>
<td>instant</td>
<td>instant</td>
</tr>
<tr>
<td></td>
<td>1 second</td>
<td>instant</td>
</tr>
<tr>
<td></td>
<td>1 week</td>
<td>instant</td>
</tr>
</tbody>
</table>

Bottom line. Good algorithms are better than supercomputers.
Mergesort: mathematical analysis

**Proposition.** Mergesort uses $\sim 2N \lg N$ data moves to sort any array of size $N$.

**Def.** $D(N) =$ number of data moves to mergesort an array of size $N$.

\[
D(N) = D(N/2) + D(N/2) + 2N
\]

**Mergesort recurrence.** $D(N) = 2D(N/2) + 2N$ for $N > 1$, with $T(1) = 0$.

- Not quite right for odd $N$.
- Similar recurrence holds for many divide-and-conquer algorithms.

**Solution.** $D(N) \sim 2N \lg N$.

- For simplicity, we'll prove when $N$ is a power of 2.
- True for all $N$. [see COS 340]
Mergesort recurrence: proof 2

**Mergesort recurrence.** \( D(N) = 2 \ D(N/2) + 2 \ N \) for \( N > 1 \), with \( D(1) = 0 \).

**Proposition.** If \( N \) is a power of 2, then \( D(N) = 2 \ N \lg N \).

**Pf.**

\[
\begin{align*}
D(N) &= 2 \ D(N/2) + 2N \\
\frac{D(N)}{N} &= 2 \ \frac{D(N/2)}{N} + 2 \\
&= \frac{D(N/2)}{(N/2)} + 2 \\
&= \frac{D(N/4)}{(N/4)} + 2 + 2 \\
&= \frac{D(N/8)}{(N/8)} + 2 + 2 + 2 \\
&= \cdots \\
&= \frac{D(N/N)}{N/N} + 2 + 2 + \ldots + 2 \\
&= 2 \ lg N
\end{align*}
\]
Mergesort recurrence: proof 3

**Mergesort recurrence.** \( D(N) = 2 \, D(N/2) + 2 \, N \) for \( N > 1 \), with \( D(1) = 0 \).

**Proposition.** If \( N \) is a power of 2, then \( D(N) = 2 \, N \, \lg N \).

**Pf.** [by induction on \( N \)]

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

\[
D(2N) = 2 \, D(N) + 4N \\
= 4 \, N \, \lg N + 4 \, N \\
= 4 \, N \, (\lg (2N) - 1) + 4N \\
= 4 \, N \, \lg (2N)
\]

given

inductive hypothesis

algebra

QED
Mergesort analysis: memory

**Proposition G.** Mergesort uses extra space proportional to \( N \).

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

**Def.** A sorting algorithm is **in-place** if it uses \( O(\log N) \) extra memory.

**Ex.** Insertion sort, selection sort, shellsort.

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

Use insertion sort for small subarrays.
- Mergesort has too much overhead for tiny subarrays.
- Cutoff to insertion sort for \( \approx 7 \) elements.

Stop if already sorted.
- Is biggest element in first half \( \leq \) smallest element 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.

Ex. See `MergeX.java` or `Arrays.sort()`.
Mergesort visualization

Visual trace of top-down mergesort for with cutoff for small subarrays
- mergesort
- **bottom-up mergesort**
- sorting complexity
- comparators
### Bottom-up mergesort

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

#### Trace of merge results for bottom-up mergesort

<table>
<thead>
<tr>
<th>sz</th>
<th>a[i]</th>
<th>MERGESORTEXAMPLE</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>MERGESORTEXAMPLE</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>MERGESORTEXAMPLE</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>MERGESORTEXAMPLE</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>MERGESORTEXAMPLE</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>MERGESORTEXAMPLE</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>MERGESORTEXAMPLE</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>MERGESORTEXAMPLE</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>MERGESORTEXAMPLE</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>MERGESORTEXAMPLE</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td>MERGESORTEXAMPLE</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>MERGESORTEXAMPLE</td>
</tr>
<tr>
<td>11</td>
<td>11</td>
<td>MERGESORTEXAMPLE</td>
</tr>
<tr>
<td>12</td>
<td>12</td>
<td>MERGESORTEXAMPLE</td>
</tr>
<tr>
<td>13</td>
<td>13</td>
<td>MERGESORTEXAMPLE</td>
</tr>
<tr>
<td>14</td>
<td>14</td>
<td>MERGESORTEXAMPLE</td>
</tr>
<tr>
<td>15</td>
<td>15</td>
<td>MERGESORTEXAMPLE</td>
</tr>
</tbody>
</table>

#### Bottom line.
No recursion needed!
Bottom-up mergesort: Java implementation

public class MergeBU
{
   private static Comparable[] aux;

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

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

Bottom line. Concise industrial-strength code, if you have the space.
Bottom-up mergesort: visual trace

Visual trace of bottom-up mergesort