2.2 Mergesort

- mergesort
- bottom-up mergesort
- sorting complexity
- divide-and-conquer
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
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 \( a[hi] \).

<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>
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].`

<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>

**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]; // copy from a to aux

    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++]; // merge from aux to a
    }
}
Mergesort quiz 1

How many calls does merge() make to to less() to merge two sorted subarrays of size \( \frac{N}{2} \) each into a sorted array of size \( N \).

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. \( \sim N \)
E. I don't know.
**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);
    }
}
```
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
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 analysis: empirical running time

Running time estimates:

- Laptop executes $10^8$ compares/second.
- Supercomputer executes $10^{12}$ compares/second.

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

Bottom line. Good algorithms are better than supercomputers.
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:

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

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))
**Divide-and-conquer recurrence**

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

**Pf by picture.** [assuming $N$ is a power of 2]
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 $D(2N) = (2N) \lg (2N)$.

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

substitute 2N into given formula

inductive hypothesis

algebra

QED
**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:

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

**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);  \text{solve two problems}
    f(N/2);  \text{of half the size}
    linear(N); \text{do a linear amount of work}
}
```

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

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

**Pf.** The array $\text{aux}[\cdot]$ 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, ... 

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

**Challenge 2 (very hard).** In-place merge. [Kronrod 1969]
Mergesort quiz 3

Is our implementation of mergesort stable?

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

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

<table>
<thead>
<tr>
<th>input</th>
<th>C</th>
<th>A₁</th>
<th>B</th>
<th>A₂</th>
<th>A₃</th>
</tr>
</thead>
<tbody>
<tr>
<td>sorted</td>
<td>A₃</td>
<td>A₁</td>
<td>A₂</td>
<td>B</td>
<td>C</td>
</tr>
</tbody>
</table>

don't stable
Stability: mergesort

Proposition. Mergesort is stable.

Pf. Suffices to verify that merge operation is stable.
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++];
    }
}
```

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 with cutoff to insertion sort: visualization

- first subarray
- second subarray
- first merge
- first half sorted
- second half sorted
- result
Mergesort: practical improvements

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

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

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

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

merge from aux[] to a[]
assumes aux[] is initialized to a[] once, before recursive calls
switch roles of aux[] and a[]
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.

http://hg.openjdk.java.net/jdk6/jdk6/jdk/file/tip/src/share/classes/java/util/Arrays.java
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, ....

\[
a[i] \\
\begin{array}{cccccccccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 \\
M & E & R & G & E & S & O & R & T & E & X & A & M & P & L & E \\
\end{array}
\]

\[
\begin{array}{c}
sz = 1 \\
merge(a, aux, 0, 0, 1) \\
merge(a, aux, 2, 2, 3) \\
merge(a, aux, 4, 4, 5) \\
merge(a, aux, 6, 6, 7) \\
merge(a, aux, 8, 8, 9) \\
merge(a, aux, 10, 10, 11) \\
merge(a, aux, 12, 12, 13) \\
merge(a, aux, 14, 14, 15) \\
sz = 2 \\
merge(a, aux, 0, 1, 3) \\
merge(a, aux, 4, 5, 7) \\
merge(a, aux, 8, 9, 11) \\
merge(a, aux, 12, 13, 15) \\
sz = 4 \\
merge(a, aux, 0, 3, 7) \\
merge(a, aux, 8, 11, 15) \\
sz = 8 \\
merge(a, aux, 0, 7, 15)
\]

\[
\begin{array}{c}
A & E & E & E & E & G & L & M & M & O & P & R & R & S & T & X \\
\end{array}
\]
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: top-down mergesort or bottom-up mergesort? You may assume \( N \) is a power of 2.

A. Top-down (recursive) mergesort.
B. Bottom-up (nonrecursive) mergesort.
C. About the same.
D. I don't know.
Natural mergesort

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

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

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

---

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 \( \log(N!) \) 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, ....

http://hg.openjdk.java.net/jdk7/jdk7/jdk/file/tip/src/share/classes/java/util/Arrays.java
## 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># 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>
Decision tree (for 3 distinct keys a, b, and c)

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

  - b < c
    - yes: a c b
      - yes: a c b (ordering)
      - no: c a b
    - no: a < c
      - yes: a c b (ordering)
      - no: c a b

  - a < c
    - yes: b a c
      - yes: b a c (ordering)
      - no: b c a
    - no: b < c
      - yes: b c a (ordering)
      - no: c b a

- height of tree = worst-case number of compares

- code between compares (e.g., sequence of exchanges)

- 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.

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.
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.

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

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>complexity of sorting</th>
</tr>
</thead>
<tbody>
<tr>
<td>model of computation</td>
</tr>
<tr>
<td>cost model</td>
</tr>
<tr>
<td>upper bound</td>
</tr>
<tr>
<td>lower bound</td>
</tr>
<tr>
<td>optimal algorithm</td>
</tr>
</tbody>
</table>

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

**Compared?** Mergesort is **optimal** with respect to number of compares.

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

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

**Ex.** Design sorting algorithm that guarantees \( \sim \frac{1}{2} N \log N \) 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.
2.2 Mergesort

- mergesort
- bottom-up mergesort
- sorting complexity
- divide-and-conquer
**INTERVIEW QUESTION: SORT A LINKED LIST**

**Problem.** Given a singly-linked list, rearrange its nodes in sorted order.

- Linearithmic time, logarithmic (or constant) extra space.

```
input
5 6 2 7 3 4 null

sorted
2 3 4 5 6 7 null
```