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]

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].`

---

### Merging: Java implementation

```java
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 + 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

```
Mergesort: trace

<table>
<thead>
<tr>
<th>lo</th>
<th>hi</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>7</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td>8</td>
<td>11</td>
</tr>
<tr>
<td>10</td>
<td>13</td>
</tr>
<tr>
<td>12</td>
<td>15</td>
</tr>
</tbody>
</table>
```

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

Mergesort: animation

50 random items

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

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

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

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

We solve the recurrence when $N$ is a power of 2:

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

Divide-and-conquer recurrence: proof by picture

**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 1.** [assuming $N$ is a power of 2]

Given

\[
\begin{array}{cccccccccccc}
N & - N \\
2(N/2) & - N \\
4(N/4) & - N \\
8(N/8) & - N \\
& \vdots \\
T(N) & = N\lg N
\end{array}
\]

Mergesort: 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:

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

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

```java
public static void linearithmic(int N) {
  if (N == 0) return;
  linearithmic(N/2);
  linearithmic(N/2);
  linear(N);
}
```

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

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

**Pf.** The array $\text{aux[]}$ 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 $\approx \frac{1}{2} N$ instead of $N$.

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

Mergesort with cutoff to insertion sort: visualization

---

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) 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.

```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], a[mid+1]) 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[i], a[j])) aux[k] = a[i++];
        else aux[k] = a[j++];
    }
    merge(a, aux, lo, mid, hi);
}
```

```java
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);
}
```

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.

```java
Arrays.sort(a)
```

http://www.java2s.com/Open-Source/Java/6.0-JDK-Modules/j2me/java/util/Arrays.java.html

Bottom-up mergesort

Basic plan.

- Pass through array, merging subarrays of size 1.
- Repeat for subarrays of size 2, 4, 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.

---

**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.

---

**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 sample sort 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, …
The Zen of Python

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren’t special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, reject the temptation to guess.
There should be one — and preferably only one — obvious way to do it.
Although that way may not be obvious at first unless you’re Dutch.
Now is better than never.
Although never is often better than right now.
If the implementation is hard to explain, it’s a bad sign.
If the implementation is easy to explain, it may be a good sign.
Namespaces are one honking great idea — let’s do more of those.

http://www.python.org/dev/peps/pep-0020/

2.2 Mergesort

▷ mergesort
▷ bottom-up mergesort
▷ sorting complexity
▷ comparators
▷ stability

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 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 \( X \).

Example: sorting.

▷ Model of computation: decision tree.
▷ Cost model: \# compares.
▷ Upper bound: \( \sim N \lg N \) from mergesort.
▷ Lower bound: \( \sim N \lg N \) from mergesort.

Decision tree (for 3 distinct keys \( a \), \( b \), and \( c \))

- 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)

Example:

- Model of computation: decision tree.
- Cost model: \# compares.
- Upper bound: \( \sim N \lg N \) from mergesort.
- Lower bound: \( \sim N \lg N \) from mergesort.

Optimal algorithm:

- Model of computation: decision tree.
- Cost model: \# compares.
- Upper bound: \( \sim N \lg N \) from mergesort.
- Lower bound: \( \sim N \lg N \) from mergesort.

Optimal algorithm:
**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 \( X \).

**Example: sorting.**
- Model of computation: decision tree.
- Cost model: \# compares.
- Upper bound: \( \sim N \lg N \) from mergesort.
- Lower bound: \( \sim N \lg N \).
- Optimal algorithm = mergesort.

**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 \( \frac{1}{2} N \lg N \) compares?
**Ex.** Design sorting algorithm that is both time- and space-optimal?
Complexity results in context (continued)

Lower bound may not hold if the algorithm can take advantage of:

- The initial order of the input.
  Ex: insert 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 sort requires no key compares — it accesses the data via character/digit compares.
Sort music library by artist

Comparable interface: review

Comparable interface: sort using a type's natural order.

public class Date implements Comparable<Date> {
    private final int month, day, year;

    public Date(int m, int d, int y) {
        month = m;
        day = d;
        year = y;
    }

    public int compareTo(Date that) {
        if (this.year < that.year) return -1;
        if (this.year > that.year) return +1;
        if (this.month < that.month) return -1;
        if (this.month > that.month) return +1;
        if (this.day < that.day) return -1;
        if (this.day > that.day) return +1;
        return 0;
    }
}

Comparator interface

Comparator interface: sort using an alternate order.

public interface Comparator<Key> {
    int compare(Key v, Key w)
    // compare keys v and w
}

Required property. Must be a total order.

natural order | Now is the time | pre-1994 order for digraphs ch and ll and rr
---------------|-----------------|
case insensitive | is Now the time |
Spanish language | café cafetero cuarto churró nube | café
British phone book | McKinley Mackintosh | McInley Mackintosh
Comparator interface: system sort

To use with Java system sort:
- Create Comparator object.
- Pass as second argument to Arrays.sort().

Bottom line. Decouples the definition of the data type from the definition of what it means to compare two objects of that type.

Comparator interface: implementing

To implement a comparator:
- Define a (nested) class that implements the Comparator interface.
- Implement the compare() method.

public class Student
{
    private final String name;
    private final int section;
    ...

    public static class ByName implements Comparator<Student>
    {
        public int compare(Student v, Student w)
        {
            return v.name.compareTo(w.name);
        }
    }

    public static class BySection implements Comparator<Student>
    {
        public int compare(Student v, Student w)
        {
            return v.section - w.section;
        }
    }
}

Comparator interface: using with our sorting libraries

To support comparators in our sort implementations:
- Use Object instead of Comparable.
- Pass Comparator to sort() and less() and use it in less().

insertion sort using a Comparator

```java
public static void sort(Object[] a, Comparator comparator)
{
    int N = a.length;
    for (int i = 0; i < N; i++)
    {
        for (int j = 1; j > 0 && less(comparator, a[j], a[j-1]); j--)
            exch(a, j, j-1);
    }
}
```

Comparator interface: implementing

To implement a comparator:
- Define a (nested) class that implements the Comparator interface.
- Implement the compare() method.

Arrays.sort(a, new Student.BySection());
2.2 MERGESORT

- mergesort
- bottom-up mergesort
- sorting complexity
- comparators
- stability

Selection sort

Stability

Q. Which sorts are stable?
A. Need to check algorithm (and implementation).

@#$%&! Students in section 3 no longer sorted by name.

A stable sort preserves the relative order of items with equal keys.

Stability: insertion sort

Proposition. Insertion sort is stable.

Pf. Equal items never move past each other.
**Stability: selection sort**

**Proposition.** Selection sort is **not stable**.

```
public class Selection {
    public static void sort(Comparable[] a) {
        int N = a.length;
        for (int i = 0; i < N; i++) {
            int min = i;
            for (int j = i+1; j < N; j++)
                if (less(a[j], a[min]))
                    min = j;
            exch(a, i, min);
        }
    }
}
```

Pf by counterexample. Long-distance exchange can move one equal item past another one.

**Stability: shellsort**

**Proposition.** Shellsort sort is **not stable**.

```
public class Shell {
    public static void sort(Comparable[] a) {
        int N = a.length;
        int h = 1;
        while (h < N/3) h = 3*h + 1;
        while (h >= 1) {
            for (int i = 0; i < N; i++)
                for (int j = i; j < N && less(a[j], a[j+h]); j += h)
                    exch(a, j, j-h);
            h = h/3;
        }
    }
}
```

Pf by counterexample. Long-distance exchanges.

**Stability: mergesort**

**Proposition.** Mergesort is **stable**.

```
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**.

```
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.
## Sorting summary

<table>
<thead>
<tr>
<th>Place?</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>$\frac{1}{2} N^2$</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>
</tr>
<tr>
<td>shell</td>
<td>✔️</td>
<td>$N \log_3 N$</td>
<td>?</td>
<td>$c N^{5/2}$</td>
<td>tight code; subquadratic</td>
</tr>
<tr>
<td>merge</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>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>holy sorting grail</td>
</tr>
</tbody>
</table>