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**

---

First Draft of a Report on the EDVAC

John von Neumann
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 \(a[hi]\).

---

**Mergesort: Transylvanian-Saxon folk dance**

---

**Merging: Java implementation**

```java
private static void merge(Comparable[] a, Comparable[] aux, int lo, int mid, int hi) {
    int i = lo, j = mid+1; // copy
    for (int k = lo; k <= hi; k++) // merge
        aux[k] = a[lo];
    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++];
}
```
Mergesort quiz 1

How many calls to `less()` does `merge()` make in the worst case to merge two subarrays of length \( N/2 \) into an array of length \( N \). Assume \( N \) is even.

A. \( N/2 \)
B. \( N/2 + 1 \)
C. \( N - 1 \)
D. \( N \)
E. I don’t know.

Mergesort quiz 2

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: 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: empirical analysis

Running time estimates:
- Laptop executes $10^8$ compares/second.
- Supercomputer executes $10^{12}$ compares/second.

<table>
<thead>
<tr>
<th>Insertion sort (N²)</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>

Mergesort analysis: number of compares

Proposition. Mergesort uses $\leq N \log N$ 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:

$$C(N) \leq C \left( \left\lfloor \frac{N}{2} \right\rfloor \right) + C \left( \left\lfloor \frac{N}{2} \right\rfloor \right) + N - 1 \quad \text{for } N > 1, \text{ with } C(1) = 0.$$ 

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

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

Bottom line. Good algorithms are better than supercomputers.
**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]

\[
\begin{align*}
D(N) & \quad N = N \\
D(N/2) & \quad 2(N/2) = N \\
D(N/4) & \quad 4(N/4) = N \\
D(N/8) & \quad 8(N/8) = N \\
\vdots & \\
T(N) & = N\lg N
\end{align*}
\]

**Mergesort analysis: 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:

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

**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);
    f(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.

**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.

**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]
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) {
        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.

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);
}
```

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 (k > 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 with cutoff to insertion sort: visualization
Mergesort: practical improvements

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

```
  Mergesort: practical improvements

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
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);
}
```

```
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) && (j <= hi))
            aux[k] = a[j++];
        else if (j > hi)
            aux[k] = a[i++];
        else if (less(a[i], a[j]))
            aux[k] = a[j++];
        else
            aux[k] = a[i++];
    }
```

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.

```
private static void merge(Comparable[] a, Comparable[] aux, int lo, 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[j++];
        else
            aux[k] = a[i++];
    }
```

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://algs4.cs.princeton.edu

http://hg.openjdk.java.net/jdk6/jdk6/jdk/file/tip/src/share/classes/java/util/Arrays.java
Bottom-up mergesort

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

Mergesort: visualizations

Bottom-up mergesort: Java implementation

public class MergeBU
{
    private 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 quiz 4

Which is faster in practice: top-down mergesort or bottom-up mergesort?

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 5 10 16 3 4 23 9 13 2 7 8 12 14</th>
</tr>
</thead>
<tbody>
<tr>
<td>First run</td>
<td>1 5 10 16 3 4 23 9 13 2 7 8 12 14</td>
</tr>
<tr>
<td>Second run</td>
<td>1 5 10 16 3 4 23 9 13 2 7 8 12 14</td>
</tr>
<tr>
<td>Merge two runs</td>
<td>1 3 4 5 10 16 23 9 13 2 7 8 12 14</td>
</tr>
</tbody>
</table>

**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 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

Commercial break

https://www.youtube.com/watch?v=tSEHDBSynVo

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>

**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

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

```
  a < b
     / \   |
    /   \  |
   /     \|
  a < c  b < c
     /  \   |
    /    \  |
   /     \|
  a < c  a < c
     /  \   |
    /    \  |
   /     \|
  a < c  a < c
     /  \   |
    /    \  |
   /     \|
  a < c  a < 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

\( 2^h \geq \# \text{ leaves} \geq N! \)

\( \Rightarrow h \geq \lg (N!) \sim N \lg N \)

Stirling’s formula

\[
\sim N \lg N
\]
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

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?

Complexity results in context (continued)

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.

Sorting summary

<table>
<thead>
<tr>
<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>( \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>( 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>( N \log_2 N )</td>
<td>?</td>
<td>( \sim N^{3/2} )</td>
<td>tight code; subquadratic</td>
</tr>
<tr>
<td>merge</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>( 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>( 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

**INTERVIEW QUESTION: SHUFFLE A LINKED LIST**

**Problem.** Given a singly-linked list, rearrange its nodes uniformly at random.

**Assumption.** Access to a perfect random-number generator.

**Version 1.** Linear time, linear extra space.

**Version 2.** Linearithmic time, logarithmic or constant extra space.

```plaintext
input
first
2♣ 3♣ 4♣ 5♣ 6♣ 7♣ null

shuffled
first
5♣ 6♣ 2♣ 7♣ 3♣ 4♣ null

input shuffled
all N! permutations equally likely
```