Mergesort

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.

Basic plan.

- Divide array into two halves.
- Recursively sort each half.
- Merge two halves.

Mergesort overview
Mergesort: Java implementation

```java
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, mid, hi);
        }

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

Merging

Goal. Combine two sorted subarrays into a sorted whole.

Q. How to merge efficiently?

A. Use an auxiliary array.
Mergesort visualization

Visual trace of top-down mergesort with cutoff for small subfiles.

Mergesort animation

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

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

Proposition. Mergesort uses $\sim N \log N$ compares to sort any array of size $N$.

Def. $T(N) =$ number of compares to mergesort an array of size $N$.

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

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

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

Solution. $T(N) \sim N \log N$.

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

Mergesort recurrence: proof 1

Proposition. If $N$ is a power of 2, then $T(N) = N \log N$.

Pf.

\[
\begin{align*}
T(N) &= 2T(N/2) + N \\
T(N) / N &= 2T(N/2) / N + 1 \\
&= T(N/2) / (N/2) + 1 \\
&= T(N/4) / (N/4) + 1 + 1 \\
&= T(N/8) / (N/8) + 1 + 1 + 1 \\
&\cdots \\
&= T(N/N) / (N/N) + 1 + 1 + \cdots + 1 \\
&= \log N
\end{align*}
\]

Mergesort recurrence: proof 2

Proposition. If $N$ is a power of 2, then $T(N) = N \log N$.

Pf. [by induction on $N$]

- Base case: $N = 1$.
- Inductive hypothesis: $T(N) = N \log N$.
- Goal: show that $T(2N) = 2N \log (2N)$.

\[
\begin{align*}
T(2N) &= 2T(N) + 2N \\
&= 2N \log N + 2N \\
&= 2N (\log (2N) - 1) + 2N \\
&= 2N \log (2N)
\end{align*}
\]

Mergesort recurrence: proof 3

Proposition. If $N$ is a power of 2, then $T(N) = N \log N$.

Pf. [by induction on $N$]

- Base case: $N = 1$.
- Inductive hypothesis: $T(N) = N \log N$.
- Goal: show that $T(2N) = 2N \log (2N)$.
Mergesort analysis: memory

Proposition G. Mergesort uses extra space proportional to N.

Pf. The array 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 ≈ 7 elements.

Stop if already sorted.
- Is biggest element in first half ≤ smallest element in second half?
- Helps for nearly ordered lists.

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 Arrays.sort().

Bottom-up mergesort

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

Bottom line. No recursion needed!
public class MergeBU
{
    private static Comparable[] aux;

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

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

Bottom line. Concise industrial-strength code, if you have the space.

Visual trace of bottom-up mergesort
Complexity of sorting

**Computational complexity.** Framework to study efficiency of algorithms for solving a particular problem X.

**Machine model.** Focus on fundamental operations.

**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 cost guarantee for X.

Example: sorting.
- Machine model = # compares.
- Upper bound = \( \sim N \lg N \) from mergesort.
- Lower bound = \( \sim N \lg N \)?
- Optimal algorithm = mergesort?

Compare-based lower bound for sorting

**Proposition.** Any compare-based sorting algorithm must use more than \( N \lg N - 1.44 N \) comparisons in the worst-case.

**Pf.**
- Assume input 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 more than \(N \lg N - 1.44N\) comparisons in the worst-case.

**Pf.**
- Assume input 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.

\[
2^h \geq N!
\]
\[
h \geq \lg N!
\]
\[
\geq \lg \left(\frac{N!}{e}\right) N
\]
\[
= N \lg N - N \lg e
\]
\[
\geq N \lg N - 1.44N
\]

Stirling's formula

Complexity of sorting

**Machine model.** Focus on fundamental operations.

**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 cost guarantee for \(X\).

**Example: sorting.**
- Machine model \(= \#\) compares.
- Upper bound \(\approx N \lg N\) from mergesort.
- Lower bound \(\approx N \lg N\).
- Optimal algorithm = mergesort.

First goal of algorithm design: optimal algorithms.

Complexity results in context

**Other operations?** Mergesort optimality is only about number of compares.

**Space?**
- Mergesort is not optimal with respect to space usage.
- Insertion sort, selection sort, and shellsort are space-optimal.

**Challenge.** Find an algorithm that is both time- and space-optimal.

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

**Ex.** Don’t try to design sorting algorithm that uses \(\frac{1}{2} N \lg N\) compares.

Complexity results in context (continued)

**Lower bound may not hold if the algorithm has information about:**
- The key values.
- Their initial arrangement.

**Partially ordered arrays.** Depending on the initial order of the input, we may not need \(N \lg N\) compares.

**Duplicate keys.** Depending on the input distribution of duplicates, we may not need \(N \lg N\) compares.

**Digital properties of keys.** We can use digit/character compares instead of key compares for numbers and strings.
sort uses type's natural order...

Generalized compare

Comparable interface: sort uses type's natural order.

Problem 1. May want to use a non-natural order.

Problem 2. Desired data type may not come with a "natural" order.

Ex. Sort strings by:

- Natural order.
- Case insensitive.
- Spanish.
- British phone book.

```java
String[] a;...
Arrays.sort(a);
Arrays.sort(a, String.CASE_INSENSITIVE_ORDER);
Arrays.sort(a, Collator.getInstance(Locale.SPANISH));
```

Comparators

Solution. Use Java's Comparator interface.

```java
public interface Comparator<Key>
{
    public int compare(Key v, Key w);
}
```

Remark. The compare() method implements a total order like compareTo().

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

- Can add any number of new orders to a data type.
- Can add an order to a library data type with no natural order.
Comparator example

Reverse order. Sort an array of strings in reverse order.

```java
public class ReverseOrder implements Comparator<String> {
    public int compare(String a, String b) {
        return b.compareTo(a);
    }
}
```

Comparator implementation

```java
Arrays.sort(a, new ReverseOrder());
```

client

Sort implementation with comparators

To support comparators in our sort implementations:
- Pass Comparator to sort() and less().
- Use it in less().

Ex. Insertion sort.

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

```java
private static <Key> boolean less(Comparator<Key> c, Key v, Key w) {
    return c.compare(v, w) < 0;
}
```

```java
private static <Key> void exch(Key[] a, int i, int j) {
    Key swap = a[i];
    a[i] = a[j];
    a[j] = swap;
}
```

pedantic Java code (see book for simpler version)

Generalized compare

Comparators enable multiple sorts of a single file (by different keys).

Ex. Sort students by name or by section.

```java
Arrays.sort(students, Student.BY_NAME);
Arrays.sort(students, Student.BY_SECT);
```

sort by name

<table>
<thead>
<tr>
<th>Name</th>
<th>Phone</th>
<th>Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>Andrews</td>
<td>664-480-0023</td>
<td>097 Little</td>
</tr>
<tr>
<td>Battle</td>
<td>874-088-1212</td>
<td>121 Whitman</td>
</tr>
<tr>
<td>Chen</td>
<td>991-878-4944</td>
<td>308 Blair</td>
</tr>
<tr>
<td>Fox</td>
<td>884-232-5341</td>
<td>112 Winston</td>
</tr>
<tr>
<td>Furia</td>
<td>884-232-5341</td>
<td>112 Winston</td>
</tr>
<tr>
<td>Gazsi</td>
<td>898-232-5341</td>
<td>112 Winston</td>
</tr>
<tr>
<td>Kanaga</td>
<td>898-232-5341</td>
<td>112 Winston</td>
</tr>
<tr>
<td>Rohde</td>
<td>898-232-5341</td>
<td>112 Winston</td>
</tr>
</tbody>
</table>

then by section

<table>
<thead>
<tr>
<th>Name</th>
<th>Phone</th>
<th>Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>Andrews</td>
<td>664-480-0023</td>
<td>097 Little</td>
</tr>
<tr>
<td>Battle</td>
<td>874-088-1212</td>
<td>121 Whitman</td>
</tr>
<tr>
<td>Chen</td>
<td>991-878-4944</td>
<td>308 Blair</td>
</tr>
<tr>
<td>Fox</td>
<td>884-232-5341</td>
<td>112 Winston</td>
</tr>
<tr>
<td>Furia</td>
<td>884-232-5341</td>
<td>112 Winston</td>
</tr>
<tr>
<td>Gazsi</td>
<td>898-232-5341</td>
<td>112 Winston</td>
</tr>
<tr>
<td>Kanaga</td>
<td>898-232-5341</td>
<td>112 Winston</td>
</tr>
<tr>
<td>Rohde</td>
<td>898-232-5341</td>
<td>112 Winston</td>
</tr>
</tbody>
</table>

Generalized compare

Ex. Enable sorting students by name or by section.

```java
public class Student {
    public static final Comparator<Student> BY_NAME = new ByName();
    public static final Comparator<Student> BY_SECT = new BySect();
    private final String name;
    private final int section;
    ...
    private static class ByName implements Comparator<Student> {
        public int compare(Student a, Student b) {
            return a.name.compareTo(b.name);
        }
    }
    private static class BySect implements Comparator<Student> {
        public int compare(Student a, Student b) {
            return a.section - b.section;
        }
    }
}
```

only use this trick if no danger of overflow
Generalized compare problem

A typical application. First, sort by name; then sort by section.

Arrays.sort(students, Student.BY_NAME);

Arrays.sort(students, Student.BY_SECT);

sort students

// Students in section 3 no longer in order by name.

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

Sorting challenge 5A

Is insertion sort stable?

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

Sorting challenge 5B

Is selection sort stable?

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);
        }
    }
}
Sorting challenge 5C

Is shellsort stable?

```java
public class Shell{
    // Shellsort.
    public static void sort(Comparable[] a){
        // Sort a[] into increasing order.
        int N = a.length;
        int h = 1;
        while (h < N/3) h = 3*h + 1; // 1, 4, 13, 40, 121, 364, 1093, ...
        while (h >= 1)
            { // h-sort the file.
                for (int i = h; i < N; i++)
                    { // Insert a[i] among a[i-h], a[i-2*h], a[i-3*h]...
                        for (int j = i; j > 0 && less(a[j], a[j-h]); j -= h)
                            exch(a, j, j-h);
                    }
                h = h/3;
            }
    }
}
```

Sorting challenge 5D

Is mergesort stable?

```java
public class Merge{
    // Merge a[l..m] with a[m+1..hi].
    for (int k = lo; k < hi; k++)
        aux[k] = a[k];
    int i = lo, j = mid;
    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++];
    }
}
```

Sorting challenge 5D (continued)

Is merge stable?

```java
public static void merge(Comparable[] a, int lo, int mid, int hi){
    // Merge a[lo..m] with a[m+1..hi].
    for (int k = lo; k < hi; k++)
        aux[k] = a[k];
    int i = lo, j = mid;
    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++];
}
```

Postscript: Optimizing mergesort (a short history)

Goal: remove instructions from the inner loop.
Postscript: Optimizing mergesort (a short history)

Idea 1 (1960s): Use sentinels

```java
int a[M] := maxint; b[N] := maxint;
for (int i = 0, j = 0, k = 0; k < M+1; k++)
    if (less(aux[j], aux[i])) aux[k] = a[i++];
    aux[k] = b[j++];
```

Problem 1: Still need copy
Problem 2: No good place to put sentinels
Problem 3: Complicates data-type interface (what is infinity for your type?)

Postscript: Optimizing mergesort (a short history)

Idea 2 (1980s): Reverse copy

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

Problem: Copy still in inner loop.

Postscript: Optimizing mergesort (a short history)

Idea 3 (1990s): Eliminate copy with recursive argument switch

```java
int m = (r+l)/2;
mergesortABr(b, a, l, m);
mergesortABr(b, a, m+1, r);
mergeAB(a, l, b, l, m, b, m+1, r);
```

Problem: Complex interactions with reverse copy
Solution: Go back to sentinels.

Arrays.sort()