2.2 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 example

```
input  MERGESORT EXAMPLE
sort left half  E E G M O R R S T
sort right half  A E E L M P T X
merge results  A E E E E G L M O P R R S T X
```

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

```
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
}
```

```
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);
    }
}
```

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.

```
assert isSorted(a, lo, hi);
```

Can enable or disable at runtime. ⇒ No cost in production code.

```
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).
**Mergesort trace**

Trace of merge results for top-down mergesort

```
<table>
<thead>
<tr>
<th>l0</th>
<th>hi</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th>a[]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td></td>
</tr>
</tbody>
</table>

Mergesort trace
```

result after recursive call

**Mergesort animation**

50 random elements

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

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>insertion sort (N²)</th>
<th>mergesort (N log N)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<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>

|                      | thousand            | million         | billion        |
| super               | instant             | instant         | instant        |

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

**Proposition.** Mergesort uses $2N \lg N$ 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 $D(1) = 0$.

- 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 1

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

**Pf.**

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

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

\[
= D(N/4) / (N/4) + 2 + 2
\]

\[
= D(N/8) / (N/8) + 2 + 2 + 2
\]

\[
\ldots
\]

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

\[
= 2 \lg N
\]

given

divide both sides by $N$

algebra

apply to first term

apply to first term again

stop applying, $T(1) = 0$

### Mergesort recurrence: proof 2

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

**Pf.** [by induction on $N$]

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

\[
D(2N) = 2D(N) + 4N
\]

\[
= 4N \lg N + 4N
\]

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

\[
= 4N \lg (2N)
\]

given

inductive hypothesis

algebra

QED

### Mergesort recurrence: proof 3

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

**Pf.** [by induction on $N$]

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

\[
D(2N) = 2D(N) + 4N
\]

\[
= 4N \lg N + 4N
\]

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

\[
= 4N \lg (2N)
\]

given

inductive hypothesis

algebra

QED
Mergesort: number of compares

**Proposition.** Mergesort uses between \( \frac{1}{2} N \log N \) and \( N \log N \) compares to sort any array of size \( N \).

**Pf.** The number of compares for the last merge is between \( \frac{1}{2} N \log N \) and \( N \log N \).

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
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!

Bottom-up mergesort: Java implementation

```java
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
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 = \( N \lg N \) from mergesort.
- Lower bound = \( \sim N \lg N \)?
- Optimal algorithm = mergesort?

Complexity of sorting

Decision tree (for 3 distinct elements)

Proposition. Any compare-based sorting algorithm must use at least \( \lg N! \sim N \lg N \) compares 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 at least \( \lg N! \sim N \lg N \) compares 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.

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

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**: \( \sim N \lg N \) from mergesort.
- **Lower bound**: \( \sim N \lg N \).
- **Optimal algorithm**: mergesort.

First goal of algorithm design: optimal algorithms.

Complexity results in context

**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 initial order of the input.
- The distribution of key values.
- The representation of the keys.

**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.
mergesort
down-up mergesort
sorting complexity
comparators

Sort by artist name

Sort by song name

Natural order

Comparable interface: sort uses type's natural order.

```java
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;
    }
}
```
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.

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

Comparator example

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

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

comparator implementation

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

Sort implementation with comparators

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

Ex. Insertion sort.

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

private static boolean less(Comparator c, Object v, Object w)
{  return c.compare(v, w) < 0;   }

private static void exch(Object[] a, int i, int j)
{  Object swap = a[i]; a[i] = a[j]; a[j] = swap;  }

Comparators

Solution. Use Java’s comparator interface.

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.
### Generalized compare

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

**Ex.** Sort students by name or by section.

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

### Generalized compare problem

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

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

©#%&@!! Students in section 3 no longer in order by name.

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

### Sorting challenge 5

**Q.** Which sorts are stable?

**Insertion sort?** Selection sort? Shellsort? Mergesort?

<table>
<thead>
<tr>
<th>Student</th>
<th>Phoenix</th>
<th>Seattle</th>
<th>Chicago</th>
</tr>
</thead>
<tbody>
<tr>
<td>Battle</td>
<td>09:00:13</td>
<td>09:21:05</td>
<td>09:19:46</td>
</tr>
<tr>
<td>Chen</td>
<td>09:00:59</td>
<td>09:19:32</td>
<td>09:19:32</td>
</tr>
<tr>
<td>Fox</td>
<td>09:35:21</td>
<td>09:21:05</td>
<td>09:19:46</td>
</tr>
<tr>
<td>Furia</td>
<td>09:14:25</td>
<td>09:00:00</td>
<td>09:19:32</td>
</tr>
<tr>
<td>Gazsi</td>
<td>09:01:10</td>
<td>09:00:00</td>
<td>09:19:46</td>
</tr>
<tr>
<td>Kanaga</td>
<td>09:01:00</td>
<td>09:00:00</td>
<td>09:19:32</td>
</tr>
<tr>
<td>Rohde</td>
<td>09:19:32</td>
<td>09:00:00</td>
<td>09:19:46</td>
</tr>
<tr>
<td>Student</td>
<td>Phoenix</td>
<td>Seattle</td>
<td>Chicago</td>
</tr>
<tr>
<td>Battle</td>
<td>09:00:13</td>
<td>09:21:05</td>
<td>09:19:46</td>
</tr>
<tr>
<td>Chen</td>
<td>09:00:59</td>
<td>09:19:32</td>
<td>09:19:32</td>
</tr>
<tr>
<td>Fox</td>
<td>09:35:21</td>
<td>09:21:05</td>
<td>09:19:46</td>
</tr>
<tr>
<td>Furia</td>
<td>09:14:25</td>
<td>09:00:00</td>
<td>09:19:32</td>
</tr>
<tr>
<td>Gazsi</td>
<td>09:01:10</td>
<td>09:00:00</td>
<td>09:19:46</td>
</tr>
<tr>
<td>Kanaga</td>
<td>09:01:00</td>
<td>09:00:00</td>
<td>09:19:32</td>
</tr>
<tr>
<td>Rohde</td>
<td>09:19:32</td>
<td>09:00:00</td>
<td>09:19:46</td>
</tr>
</tbody>
</table>

### Ex. Enable sorting students by name or by section.

```java
private static class BySect implements Comparator<Student>
{
  public int compare(Student a, Student b)
  {
    return a.section - b.section;
  }
}
```

```java
public static final Comparator<Student> BY_SECT = new BySect();
```

```java
public static final Comparator<Student> BY_NAME = new ByName();
```

### Sorting challenge 5

**Q.** Which sorts are stable?

**Insertion sort?** Selection sort? Shellsort? Mergesort?
Sorting challenge 5A

Q. Is insertion sort stable?

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

A. Yes, equal elements never move past each other.

Sorting challenge 5B

Q. Is selection sort 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);
        }
    }
}

A. No, long-distance exchanges might move left element to the right of some equal element.

Sorting challenge 5C

Q. Is shellsort 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 = h; i < N; i++)
            {
                for (int j = i; j > h && less(a[j], a[j-h]); j -= h)
                    exch(a, j, j-h);
            }
            h = h/3;
        }
    }
}

A. No, long-distance exchanges.
Q. Is mergesort stable?

A. Yes, if merge is stable.

Q. Which sorts are stable?

Yes. Insertion sort, mergesort.
No. Selection sort, shellsort.

Note. Need to carefully check code ("less than" vs "less than or equal").
**Postscript: optimizing mergesort (a short history)**

**Goal.** Remove instructions from the inner loop.

```java
private static void merge(Comparable[] a, int lo, int mid, int hi) {
    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++];
}
```

**Postscript: Optimizing mergesort (a short history)**

**Idea 1 (1960s). Use sentinels.**

**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?)

```java
int mid = (lo+hi)/2;
mergesortABr(b, a, lo, mid);
mergesortABr(b, a, mid+1, r);
mergeAB(a, lo, b, lo, mid, b, mid+1, hi);
```

**Postscript: Optimizing mergesort (a short history)**

**Idea 2 (1980s). Reverse copy.**

```java
private static void merge(Comparable[] a, int lo, int mid, int hi) {
    for (int i = lo; i <= mid; i++)
        aux[i] = a[i];

    for (int j = mid+1; j <= hi; j++)
        aux[j] = a[hi-j+mid+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.**

**Problem.** Complex interactions with reverse copy.

**Solution.** Go back to sentinels.
Sorting challenge 6

Recursive argument switch is out (recommended only for pros).

Q. Why not use reverse array copy?

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