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, C++ stable sort, Python stable sort, Firefox JavaScript, ...

Quicksort
- Java sort for primitive types.
- C qsort, Unix, Visual C++, Python, Matlab, Chrome JavaScript, ...

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

Mergesort overview

```plaintext
<table>
<thead>
<tr>
<th>input</th>
<th>MERGESORT EXAMPLE</th>
</tr>
</thead>
<tbody>
<tr>
<td>sort left half</td>
<td>E E G M O R R S T</td>
</tr>
<tr>
<td>sort right half</td>
<td>E E G M O R R S A E E L M P T X</td>
</tr>
<tr>
<td>merge results</td>
<td>A E E E E E G L M O P R R S T X</td>
</tr>
</tbody>
</table>
```

First Draft of a Report on the EDVAC

- Ada von Neumann
Merging

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

Merging: Java implementation

```java
private static void merge(Comparable[] a, Comparable[] aux, int lo, int mid, int hi) {
    // precondition: a[lo..mid] sorted
    assert isSorted(a, lo, mid);
    // precondition: a[mid+1..hi] sorted
    assert isSorted(a, mid+1, 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++];
    assert isSorted(a, lo, hi);  // postcondition: a[lo..hi] sorted
}
```

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.

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

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

```java
java -ea MyProgram // enable assertions
data -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: Java implementation

```java
public class Merge {
    private static void merge(Comparable[] a, Comparable[] aux, int lo, int mid, int hi) {
        // 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) {
        aux = new Comparable[a.length];
        sort(a, aux, 0, a.length - 1);
    }
}
```
Mergesort trace

Trace of merge results for top-down mergesort

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

Mergesort animation

50 random elements

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

Mergesort animation

50 reverse-sorted elements

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>Computer</th>
<th>Insertion sort ($N^2$)</th>
<th>Mergesort ($N \log N$)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>computer</td>
<td>thousand</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: number of compares and array accesses

**Proposition.** Mergesort uses at most $N \lg N$ compares and $6N \lg N$ array accesses to sort any array of size $N$.

**Pf sketch.** The number of compares $C(N)$ and array accesses $A(N)$ to mergesort an array of size $N$ satisfies the recurrences:

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

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

We solve the simpler divide-and-conquer recurrence when $N$ is a power of 2.

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

result holds for all $N$ (see COS 340)

---

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]

\[
\begin{align*}
D(N) & = 2D(N/2) + N \\
\frac{D(N)}{N} & = 2\frac{D(N/2)}{N} + \frac{N}{N} = 2\frac{D(N/2)}{N/2} + 1 \\
& = 2\frac{D(N/4)}{N/4} + 1 + 1 \\
& = 2\frac{D(N/8)}{N/8} + 1 + 1 + 1 \\
& \quad \vdots \\
& = 2\frac{D(N/N)}{N/N} + 1 + 1 + \ldots + 1 \\
& = \lg N
\end{align*}
\]

---

Divide-and-conquer recurrence: proof by expansion

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

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

given

\[
\frac{D(N)}{N} = 2\frac{D(N/2)}{N} + \frac{N}{N} = 2\frac{D(N/2)}{N/2} + 1
\]

divide both sides by $N$

\[
\frac{D(N/2)}{N/2} = \frac{D(N/4)}{N/4} + 1 + 1
\]

algebra

\[
\frac{D(N/4)}{N/4} = \frac{D(N/8)}{N/8} + 1 + 1 + 1
\]

apply to first term

\[
\frac{D(N/8)}{N/8} = \frac{D(N/16)}{N/16} + 1 + 1 + 1 + 1
\]

apply to first term again

\[
\vdots
\]

\[
\frac{D(N/N)}{N/N} = 1 + 1 + \ldots + 1
\]

stop applying, $D(1) = 0$

\[
\lg N
\]

---

Divide-and-conquer recurrence: proof by induction

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

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

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

given

inductive hypothesis

algebra

QED
Mergesort analysis: memory

Proposition. The memory aux[] needs to be of size for last merge.

Proposition. Mergesort uses extra space proportional to N.

Pf. The array aux[] needs to be of size 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. [Kronrod, 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 the 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.
Mergesort: practical improvements

Use insertion sort for small subarrays.
- Mergesort has too much overhead for tiny subarrays.
- Cutoff to insertion sort for \( n \approx 7 \) 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!
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.

Complexity of sorting

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

Model of computation. Specify allowable operations.

Cost 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.
• Model of computation: decision tree.
• Cost model: # compares.
• Upper bound: ~ \( N \lg N \) from mergesort.
• Lower bound: ~ \( N \lg N \) ???
• Optimal algorithm: mergesort ???
**Decision tree (for 3 distinct elements a, b, and c)**

![Decision tree diagram](image)

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

\[
2^h \geq N! \Rightarrow h \geq \lg (N!) \approx N \lg N
\]

Stirling's formula

**Compare-based lower bound for sorting**

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

**Complexity of sorting**

**Model of computation.** Specify allowable operations.

**Cost 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.**
- 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

Other operations? Mergesort is optimal with respect to number of compares (e.g., but not to number of array accesses).

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. [stay tuned]

Lessons. Use theory as a guide.
Ex. Don’t try to design sorting algorithm that guarantees \( \frac{1}{2} N \lg N \) 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.

Sort by artist name
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;
    }
}
```

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:
  - Now is the time
  - Is Now the time
- Case insensitive:
  - café, cafetero, cuarto, churro, nube, año
- Spanish:
  - café, cafetero, cuarto, churro, nube, año
- British phone book:
  - McKinley, Mackintosh

```java
import java.text.Collator;
String[] a;
...
Arrays.sort(a);
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. `compare()` must implement 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);
    }
}
```

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

Comparator implementation

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

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

Sort implementation with comparators

To support comparators in our sort implementations:

- Use `Object` instead of `Comparable`.
- Pass `Comparator` to `sort()` and `less()`.
- Use it in `less()`.

Ex. Insertion sort.

```java
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);
    }
}
```

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

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

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);
```

```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;
        }
    }
}
```

Generalized compare

Ex. Enable sorting students by name or by section.

```java
public class Student
{
    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;
        }
    }
}
```

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

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

Students in section 3 no longer in order by name.

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

Q. Which sorts are stable?
Insertion sort? Selection sort? Shellsort? Mergesort?

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

Sorted by time
Sorted by location (not stable)
Sorted by location (stable)

Stability when sorting on a second key

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 more past each other.
Q. Is selection sort stable?

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

Q. Is shellsort stable?

A. No. Long-distance exchanges.

Q. Is mergesort stable?

A. Yes, if merge is stable.
Q. Is merge stable?

A. Yes, if implemented carefully (take from left subarray if equal).

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

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

```
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++];
    }
}
```

**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?)

---

**Sorting challenge 5 (summary)**

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 to").
Postscript: optimizing mergesort (a short history)


```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.

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++];
}
```

A. It is not stable (!)

Solution. Back to the standard algorithm!

Postscript: optimizing mergesort (a short history)


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

Problem. Complex interactions with reverse copy.

Solution. Go back to sentinels.

Arrays.sort()