### 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**

**input**

<table>
<thead>
<tr>
<th>M</th>
<th>E</th>
<th>R</th>
<th>G</th>
<th>E</th>
<th>S</th>
<th>O</th>
<th>R</th>
<th>T</th>
<th>E</th>
<th>X</th>
<th>A</th>
<th>M</th>
<th>P</th>
<th>L</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>E</td>
<td>G</td>
<td>M</td>
<td>O</td>
<td>R</td>
<td>R</td>
<td>S</td>
<td>T</td>
<td>E</td>
<td>X</td>
<td>A</td>
<td>M</td>
<td>P</td>
<td>L</td>
<td>E</td>
</tr>
</tbody>
</table>

**sort left half**

| E | E | G | M | O | R | R | S | T | E | X | A | E | L | M | P | T | X |

**sort right half**

| A | E | E | E | G | L | M | O | P | R | R | S | T | X |

**merge results**

| A | E | E | E | G | L | M | O | P | R | R | S | T | X |
Mergesort trace

Mergesort trace

Merging: Java implementation

```java
public static void merge(Comparable[] a, int lo, int mid, int hi)
{
    assert isSorted(a, lo, mid);    // ...                    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.

**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).
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);
    }
}

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 \( \sim N \lg N \) compares to sort any array of size \( N \).

Deﬁnition. \( T(N) = \) 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. \( T(N) \sim N \lg N \).

• 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 \( T(N) = N \lg N \).

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

\[
T(N) / N = 2 T(N/2) / N + 1
\]

divide both sides by \( N \)
apply to ﬁrst term
apply to ﬁrst term again
stop applying, \( T(1) = 0 \)

Given
\[
N = N
\]

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

\[
4 (N/4) = N
\]

\[
2^k (N/2^k) = N
\]

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

\[
N \lg N
\]

QED

Mergesort recurrence: proof 2

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

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

Pf. [by induction on \( N \)]

• Base case: \( N = 1 \).
• Inductive hypothesis: \( T(N) = N \lg N \).
• Goal: show that \( T(2N) = 2N \lg (2N) \).

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

given
inductive hypothesis
algebra

\[
= 2 N \lg N + 2 N
\]

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

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

QED

Mergesort recurrence: proof 3
**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 \( \approx 7 \) 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**

Visual trace of top-down mergesort for with cutoff for small subarrays.
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;
    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));
            }
        }
    }
    private static void merge(Comparable[] a, int lo, int mid, int hi) {
        /* as before */
    }
}
```

**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 = ~ N lg N?
• Optimal algorithm = mergesort?

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.

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.

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 = \( \sim N \log N \) from mergesort.
- Lower bound = \( \sim N \log N \).
- 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 \( \frac{1}{2} N \log N \) 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 \( N \log N \) compares.

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

Digital properties of keys. We can use digit/character compares instead of key compares for numbers and strings.

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

```java
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;
    }
}
```

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.

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 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;  }
}
```

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());
   ...
```

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);
```

```
Andrews  3  A  664-480-0023  097 Little
Battle    4  C  874-088-1212  121 Whitman
Chen      2  A  991-878-4944  308 Blair
Fox       1  A  884-232-5341  11 Dickinson
Furia     3  A  766-093-9873  101 Brown
Gazsi     4  B  665-303-0266  22 Brown
Kanaga    3  B  898-122-9643  22 Brown
Rohde     3  A  232-343-5555  343 Forbes
Gazsi     4  B  665-303-0266  22 Brown
```
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.

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>sorted by time</th>
<th>sorted by location (not stable)</th>
<th>sorted by location (stable)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chicago 09:00:00</td>
<td>Chicago 09:25:52</td>
<td>Chicago 09:00:00</td>
</tr>
<tr>
<td>Phoenix 09:00:03</td>
<td>Chicago 09:03:13</td>
<td>Chicago 09:00:00</td>
</tr>
<tr>
<td>Houston 09:00:13</td>
<td>Chicago 09:01:33</td>
<td>Chicago 09:00:00</td>
</tr>
<tr>
<td>Chicago 09:00:59</td>
<td>Chicago 09:19:46</td>
<td>Chicago 09:19:32</td>
</tr>
<tr>
<td>Chicago 09:01:10</td>
<td>Chicago 09:19:32</td>
<td>Chicago 09:19:46</td>
</tr>
<tr>
<td>Chicago 09:03:13</td>
<td>Chicago 09:00:00</td>
<td>Chicago 09:21:05</td>
</tr>
<tr>
<td>Chicago 09:00:13</td>
<td>Chicago 09:03:13</td>
<td>Chicago 09:01:10</td>
</tr>
<tr>
<td>Chicago 09:19:32</td>
<td>Houston 09:00:13</td>
<td>Houston 09:10:10</td>
</tr>
<tr>
<td>Chicago 09:19:46</td>
<td>Phoenix 09:37:44</td>
<td>Phoenix 09:00:03</td>
</tr>
<tr>
<td>Chicago 09:21:05</td>
<td>Phoenix 09:00:03</td>
<td>Phoenix 09:14:25</td>
</tr>
<tr>
<td>Seattle 09:22:54</td>
<td>Seattle 09:10:25</td>
<td>Seattle 09:10:11</td>
</tr>
<tr>
<td>Chicago 09:23:52</td>
<td>Seattle 09:36:14</td>
<td>Seattle 09:10:25</td>
</tr>
<tr>
<td>Seattle 09:36:14</td>
<td>Seattle 09:10:11</td>
<td>Seattle 09:22:54</td>
</tr>
<tr>
<td>Phoenix 09:37:44</td>
<td>Seattle 09:22:54</td>
<td>Seattle 09:36:14</td>
</tr>
</tbody>
</table>

Stability when sorting on a second key