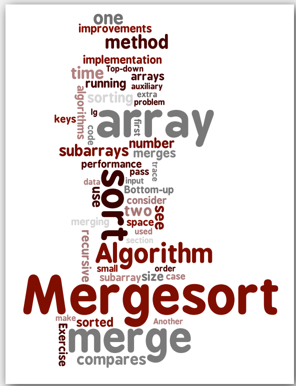
# 2.2 Mergesort




- mergesort
- bottom-up mergesort
- sorting complexity
- comparators



## 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. ← today

- Java sort for objects.
- Perl, Python stable sort.

Quicksort. ← next lecture

- Java sort for primitive types.
- C qsort, Unix, g++, Visual C++, Python.



- mergesort
- bottom-up mergesort
- sorting complexity
- comparators



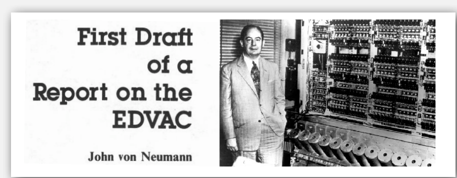
## Mergesort

Basic plan.

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

| | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| input | M | E | R | G | E | S | O | R | T | E | X | A | M | P | L | E |
| sort left half | E | E | G | M | O | R | R | S | T | E | X | A | M | P | L | E |
| sort right half | E | E | G | M | O | R | R | S | A | E | E | L | M | P | T | X |
| merge results | A | E | E | E | E | G | L | M | M | O | P | R | R | S | T | X |

Mergesort overview



## Merging

Q. How to combine two sorted subarrays into a sorted whole.

A. Use an auxiliary array.

| | k | a[0] | a[1] | a[2] | a[3] | a[4] | a[5] | a[6] | a[7] | a[8] | a[9] | i | j | aux[0] | aux[1] | aux[2] | aux[3] | aux[4] | aux[5] | aux[6] | aux[7] | aux[8] | aux[9] |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| input | | E | E | G | M | R | A | C | E | R | T | | | - | - | - | - | - | - | - | - | - | - |
| copy | | E | E | G | M | R | A | C | E | R | T | | | E | E | G | M | R | A | C | E | R | T |
| | | | | | | | | | | | | 0 | 5 | | | | | | | | | | |
| | 0 | A | | | | | | | | | | 0 | 6 | E | E | G | M | R | A | C | E | R | T |
| | 1 | A | C | | | | | | | | | 0 | 7 | E | E | G | M | R | | C | E | R | T |
| | 2 | A | C | E | | | | | | | | 1 | 7 | E | E | G | M | R | | | E | R | T |
| | 3 | A | C | E | E | | | | | | | 2 | 7 | | E | G | M | R | | | E | R | T |
| | 4 | A | C | E | E | E | | | | | | 2 | 8 | | | G | M | R | | | E | R | T |
| | 5 | A | C | E | E | E | G | | | | | 3 | 8 | | | G | M | R | | | | R | T |
| | 6 | A | C | E | E | E | G | M | | | | 4 | 8 | | | | M | R | | | | R | T |
| | 7 | A | C | E | E | E | G | M | R | | | 5 | 8 | | | | | R | | | | R | T |
| | 8 | A | C | E | E | E | G | M | R | R | | 5 | 9 | | | | | | | | | R | T |
| | 9 | A | C | E | E | E | G | M | R | R | T | 6 | 10 | | | | | | | | | | T |
| merged result | | A | C | E | E | E | G | M | R | R | T | | | | | | | | | | | | |

Abstract in-place merge trace



## Merging: Java implementation

```
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++)                                  copy
      aux[k] = a[k];

   int i = lo, j = mid+1;
   for (int k = lo; k <= hi; k++)
   {
      if      (i > mid)              a[k] = aux[j++];              merge
      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 ture.

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

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

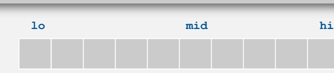
```
java -ea MyProgram   // enable assertions
java -da MyProgram   // disable assertions (default)
```

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: Java implementation

```
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);
   }
}
```



## Mergesort trace

| | a[] | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| lo / hi | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
| | M | E | R | G | E | S | O | R | T | E | X | A | M | P | L | E |
| merge(a, 0, 0, 1) | E | M | R | G | E | S | O | R | T | E | X | A | M | P | L | E |
| merge(a, 2, 2, 3) | E | M | G | R | E | S | O | R | T | E | X | A | M | P | L | E |
| merge(a, 0, 1, 3) | E | G | M | R | E | S | O | R | T | E | X | A | M | P | L | E |
| merge(a, 4, 4, 5) | E | G | M | R | E | S | O | R | T | E | X | A | M | P | L | E |
| merge(a, 6, 6, 7) | E | G | M | R | E | S | O | R | T | E | X | A | M | P | L | E |
| merge(a, 4, 5, 7) | E | G | M | R | E | O | R | S | T | E | X | A | M | P | L | E |
| merge(a, 0, 3, 7) | E | E | G | M | O | R | R | S | T | E | X | A | M | P | L | E |
| merge(a, 8, 8, 9) | E | E | G | M | O | R | R | S | E | T | X | A | M | P | L | E |
| merge(a, 10, 10, 11) | E | E | G | M | O | R | R | S | E | T | A | X | M | P | L | E |
| merge(a, 8, 9, 11) | E | E | G | M | O | R | R | S | A | E | T | X | M | P | L | E |
| merge(a, 12, 12, 13) | E | E | G | M | O | R | R | S | A | E | T | X | M | P | L | E |
| merge(a, 14, 14, 15) | E | E | G | M | O | R | R | S | A | E | T | X | M | P | E | L |
| merge(a, 12, 13, 15) | E | E | G | M | O | R | R | S | A | E | T | X | E | L | M | P |
| merge(a, 8, 11, 15) | E | E | G | M | O | R | R | S | A | E | E | L | M | P | T | X |
| merge(a, 0, 7, 15) | A | E | E | E | E | G | L | M | M | O | P | R | R | S | T | X |

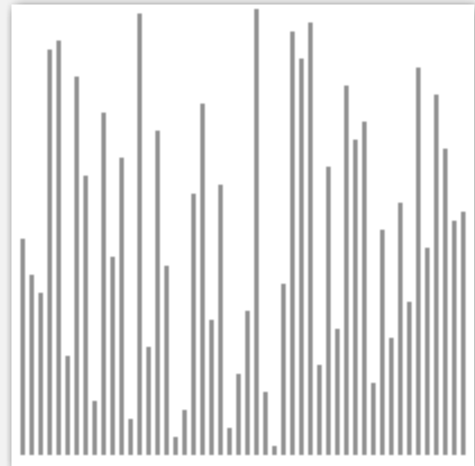
Trace of merge results for top-down mergesort

result after recursive call



## Mergesort animation

50 random elements



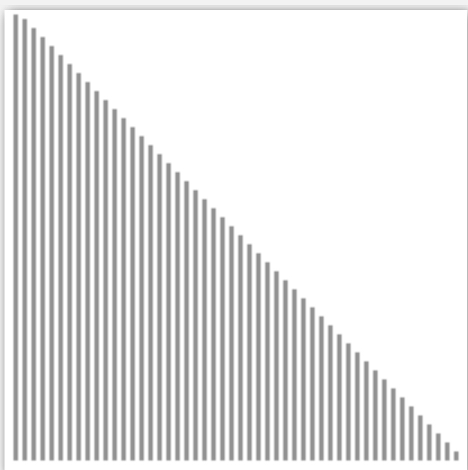
http://www.sorting-algorithms.com/merge-sort

▲ algorithm position
in order
current subarray
not in order



## Mergesort animation

50 reverse-sorted elements



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

▲ algorithm position
in order
current subarray
not in order



## Mergesort: empirical analysis

Running time estimates:

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

| | insertion sort ($N^2$) | | | mergesort (N log N) | | |
|---|---|---|---|---|---|---|
| computer | thousand | million | billion | thousand | million | billion |
| home | instant | 2.8 hours | 317 years | instant | 1 second | 18 min |
| super | instant | 1 second | 1 week | instant | instant | instant |

Bottom line. Good algorithms are better than supercomputers.

## Mergesort: mathematical analysis

Proposition. Mergesort uses $\sim 2\,N \lg N$ data moves to sort any array of size $N$.

Def. $D(N)$ = number of data moves to mergesort an array of size $N$.

$$= \underbrace{D(N/2)}_{\text{left half}} + \underbrace{D(N/2)}_{\text{right half}} + \underbrace{2\,N}_{\text{merge}}$$

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

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

Solution. $D(N) \sim 2\,N \lg N$.

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

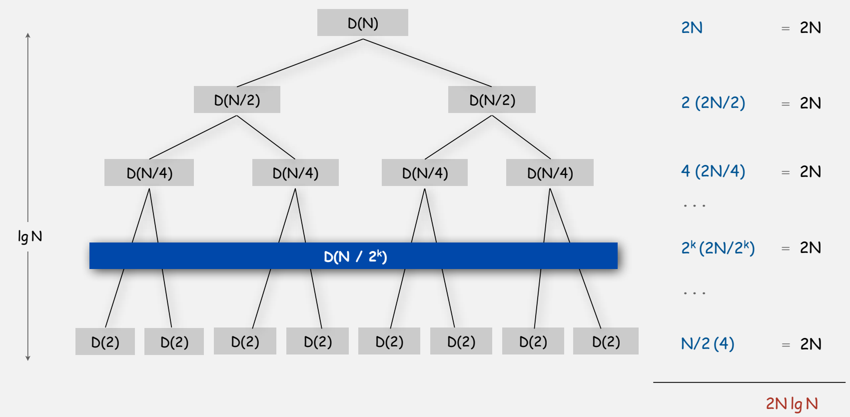


## Mergesort recurrence: proof 1

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

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

Pf.






## Mergesort recurrence: proof 2

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

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

Pf.

| | |
|---|---|
| D(N) = 2 D(N/2) + 2N | given |
| D(N) / N = 2 D(N/2) / N + 2 | divide both sides by N |
| = D(N/2) / (N/2) + 2 | algebra |
| = D(N/4) / (N/4) + 2 + 2 | apply to first term |
| = D(N/8) / (N/8) + 2 + 2 + 2 | apply to first term again |
| . . . | |
| = D(N/N) / (N/N) + 2 + 2 + ... + 2 | stop applying, T(1) = 0 |
| = 2 lg N | |



## Mergesort recurrence: proof 3

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

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

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) = 2 D(N) + 4N | given |
| = 4 N lg N + 4 N | inductive hypothesis |
| = 4 N (lg (2N) - 1) + 4N | algebra |
| = 4 N lg (2N) | QED |

## Mergesort: number of compares

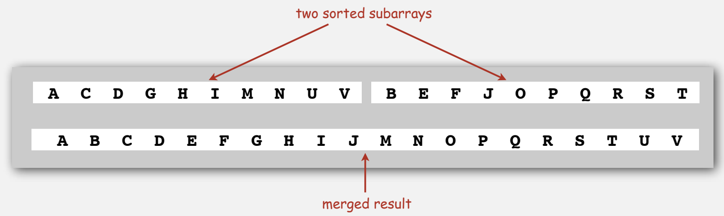
**Proposition.** Mergesort uses between $\frac{1}{2} N \lg N$ and $N \lg N$ compares to sort any array of size $N$.

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



## 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]



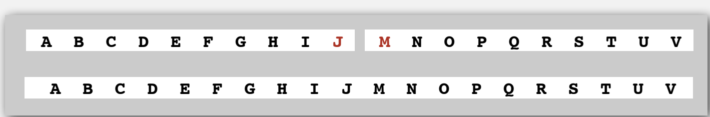
## Mergesort: practical improvements

**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 partially-ordered arrays.

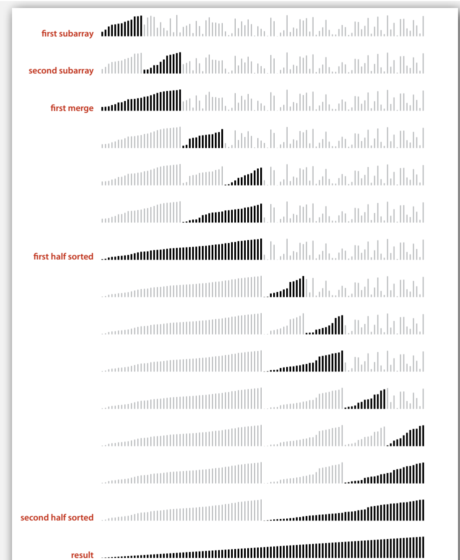



**Eliminate the copy to the auxiliary array.** Save time (but not space) by switching the role of the input and auxiliary array in each recursive call.

**Ex.** See `MergeX.java` or `Arrays.sort()`.



## Mergesort visualization




Visual trace of top-down mergesort for with cutoff for small subarrays

- mergesort
- **bottom-up mergesort**
- sorting complexity
- comparators



## Bottom-up mergesort

**Basic plan.**

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

| | a[i] 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | M | E | R | G | E | S | O | R | T | E | X | A | M | P | L | E |
| sz = 2 | | | | | | | | | | | | | | | | |
| merge(a, 0, 0, 1) | E | M | R | G | E | S | O | R | T | E | X | A | M | P | L | E |
| merge(a, 2, 2, 3) | E | M | G | R | E | S | O | R | T | E | X | A | M | P | L | E |
| merge(a, 4, 4, 5) | E | M | G | R | E | S | O | R | T | E | X | A | M | P | L | E |
| merge(a, 6, 6, 7) | E | M | G | R | E | S | O | R | T | E | X | A | M | P | L | E |
| merge(a, 8, 8, 9) | E | M | G | R | E | S | O | R | E | T | X | A | M | P | L | E |
| merge(a, 10, 10, 11) | E | M | G | R | E | S | O | R | E | T | A | X | M | P | L | E |
| merge(a, 12, 12, 13) | E | M | G | R | E | S | O | R | E | T | A | X | M | P | L | E |
| merge(a, 14, 14, 15) | E | M | G | R | E | S | O | R | E | T | A | X | M | P | E | L |
| sz = 4 | | | | | | | | | | | | | | | | |
| merge(a, 0, 1, 3) | E | G | M | R | E | S | O | R | E | T | A | X | M | P | E | L |
| merge(a, 4, 5, 7) | E | G | M | R | E | O | R | S | E | T | A | X | M | P | E | L |
| merge(a, 8, 9, 11) | E | G | M | R | E | O | R | S | A | E | T | X | M | P | E | L |
| merge(a, 12, 13, 15) | E | G | M | R | E | O | R | S | A | E | T | X | E | L | M | P |
| sz = 8 | | | | | | | | | | | | | | | | |
| merge(a, 0, 3, 7) | E | E | G | M | O | R | R | S | A | E | T | X | E | L | M | P |
| merge(a, 8, 11, 15) | E | E | G | M | O | R | R | S | A | E | E | L | M | P | T | X |
| sz = 16 | | | | | | | | | | | | | | | | |
| merge(a, 0, 7, 15) | A | E | E | E | E | G | L | M | M | O | P | R | R | S | T | X |

Trace of merge results for bottom-up mergesort

**Bottom line.** No recursion needed!



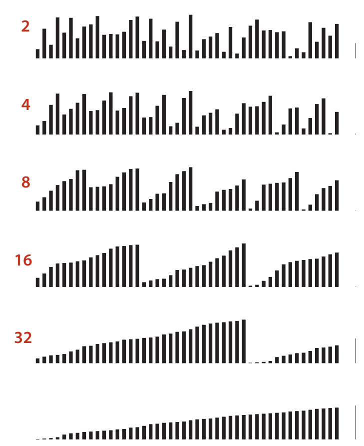
## Bottom-up mergesort: Java implementation

```
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

- mergesort
- bottom-up mergesort
- **sorting complexity**
- comparators



## 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.

lower bound ~ upper bound

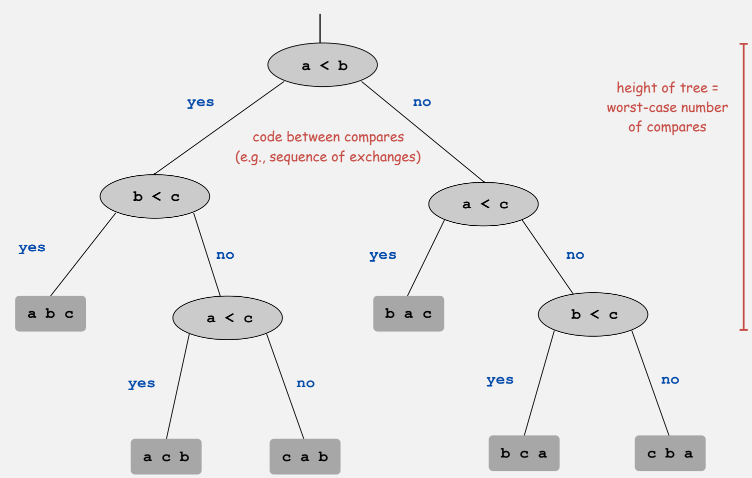
Example: sorting.

access information only through compares

- Machine model = # compares.
- Upper bound = ~ N lg N from mergesort.
- Lower bound = ~ N lg N ?
- Optimal algorithm = mergesort ?



## Decision tree (for 3 distinct elements)



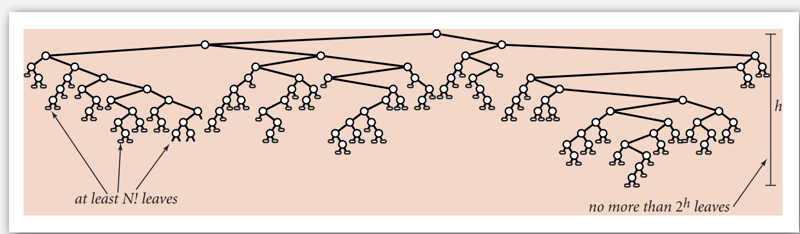



## Compare-based lower bound for sorting

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.



## 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.

$$2^h \ge \text{\# leaves} \ge N!$$
$$\Rightarrow \; h \ge \lg N! \sim N \lg N$$

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 = ~ N lg N from mergesort.
- Lower bound = ~ 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.

insertion sort requires only N-1 compares on an already sorted array

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

stay tuned for 3-way quicksort

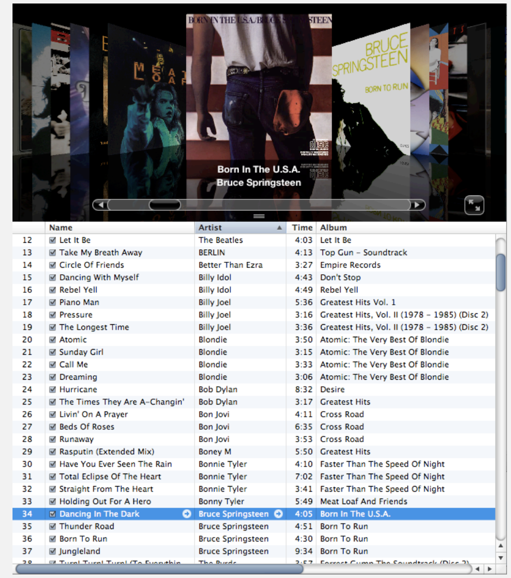
Digital properties of keys. We can use digit/character compares instead of key compares for numbers and strings.

stay tuned for radix sorts

- mergesort
- bottom-up mergesort
- sorting complexity
- **comparators**



## Sort by artist name






## Sort by song name




## Natural order

Comparable interface: sort uses type's natural order.

```
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;
   }
}
```

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. `Now is the time`
- Case insensitive. `is Now the time`
- Spanish. `café cafetero cuarto churro nube ñoño`
- British phone book. `McKinley Mackintosh`

pre-1994 order for digraphs ch and ll and rr

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

import `java.text.Collator;`



## 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.



## 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());
   ...
```

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.

```
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;  }
```

## Generalized compare

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

Ex. Sort students by name or by section.

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

sort by name

| | | | | |
|---|---|---|---|---|
| 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 |

sort by section

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



## Generalized compare

Ex. Enable sorting students by name or by section.

```
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);`

| | | | | |
|---|---|---|---|---|
| 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 |

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

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

@#%&@!!. 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?

| sorted by time | sorted by location (not stable) | sorted by location (stable) |
|---|---|---|
| Chicago 09:00:00 | Chicago 09:25:52 | Chicago 09:00:00 |
| Phoenix 09:00:03 | Chicago 09:03:13 | Chicago 09:00:59 |
| Houston 09:00:13 | Chicago 09:21:05 | Chicago 09:03:13 |
| Chicago 09:00:59 | Chicago 09:19:46 | Chicago 09:19:32 |
| Houston 09:01:10 | Chicago 09:19:32 | Chicago 09:19:46 |
| Chicago 09:03:13 | Chicago 09:00:00 | Chicago 09:21:05 |
| Seattle 09:10:11 | Chicago 09:35:21 | Chicago 09:25:52 |
| Seattle 09:10:25 | Chicago 09:00:59 | Chicago 09:35:21 |
| Phoenix 09:14:25 | Houston 09:01:10 | Houston 09:00:13 |
| Chicago 09:19:32 | Houston 09:00:13 | Houston 09:01:10 |
| Chicago 09:19:46 | Phoenix 09:37:44 | Phoenix 09:00:03 |
| Chicago 09:21:05 | Phoenix 09:00:03 | Phoenix 09:14:25 |
| Seattle 09:22:43 | Phoenix 09:14:25 | Phoenix 09:37:44 |
| Seattle 09:22:54 | Seattle 09:10:25 | Seattle 09:10:11 |
| Chicago 09:25:52 | Seattle 09:36:14 | Seattle 09:10:25 |
| Chicago 09:35:21 | Seattle 09:22:43 | Seattle 09:22:43 |
| Seattle 09:36:14 | Seattle 09:10:11 | Seattle 09:22:54 |
| Phoenix 09:37:44 | Seattle 09:22:54 | Seattle 09:36:14 |

*no longer sorted by time*

*still sorted by time*

Stability when sorting on a second key

- mergesort
- bottom-up mergesort
- sorting complexity
- comparators
- **sorting challenge**



## 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);
   }
}
```

| i | j | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|---|
| 0 | 0 | $B_1$ | $A_1$ | $A_2$ | $A_3$ | $B_2$ |
| 1 | 0 | $A_1$ | $B_1$ | $A_2$ | $A_3$ | $B_2$ |
| 2 | 1 | $A_1$ | $A_2$ | $B_1$ | $A_3$ | $B_2$ |
| 3 | 2 | $A_1$ | $A_2$ | $A_3$ | $B_1$ | $B_2$ |
| 4 | 4 | $A_1$ | $A_2$ | $A_3$ | $B_1$ | $B_2$ |
| | | $A_1$ | $A_2$ | $A_3$ | $B_1$ | $B_2$ |

A. Yes, equal elements never more 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);
      }
   }
}
```

| i | min | 0 | 1 | 2 |
|---|---|---|---|---|
| 0 | 2 | $B_1$ | $B_2$ | A |
| 1 | 1 | A | $B_2$ | $B_1$ |
| 2 | 2 | A | $B_2$ | $B_1$ |
| | | A | $B_2$ | $B_1$ |

A. No, long-distance exchange 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;
      }
   }
}
```

| h | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| | $B_1$ | $B_2$ | $B_3$ | $B_4$ | $A_1$ |
| 4 | $A_1$ | $B_2$ | $B_3$ | $B_4$ | $B_1$ |
| 1 | $A_1$ | $B_2$ | $B_3$ | $B_4$ | $B_1$ |
| | $A_1$ | $B_2$ | $B_3$ | $B_4$ | $B_1$ |

A. No. Long-distance exchanges.

## Sorting challenge 5D

**Q.** Is mergesort stable?

```
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);
   }
}
```



## Sorting challenge 5D

**Q.** Is mergesort stable?

| | a[i] | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| lo m hi | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
| | M | E | R | G | E | S | O | R | T | E | X | A | M | P | L | E |
| merge(a, 0, 0, 1) | E | M | R | G | E | S | O | R | T | E | X | A | M | P | L | E |
| merge(a, 2, 2, 3) | E | M | G | R | E | S | O | R | T | E | X | A | M | P | L | E |
| merge(a, 4, 4, 5) | E | G | M | R | E | S | O | R | T | E | X | A | M | P | L | E |
| merge(a, 6, 6, 7) | E | G | M | R | E | S | O | R | T | E | X | A | M | P | L | E |
| merge(a, 8, 8, 9) | E | E | G | M | O | R | R | S | E | T | X | A | M | P | L | E |
| merge(a, 10, 10, 11) | E | E | G | M | O | R | R | S | E | T | A | X | M | P | L | E |
| merge(a, 12, 12, 13) | E | E | G | M | O | R | R | S | A | E | T | X | M | P | L | E |
| merge(a, 14, 14, 15) | E | E | G | M | O | R | R | S | A | E | T | X | M | P | E | L |
| merge(a, 0, 1, 3) | E | G | M | R | E | S | O | R | T | E | X | A | M | P | L | E |
| merge(a, 4, 5, 7) | E | G | M | R | E | O | R | S | T | E | X | A | M | P | L | E |
| merge(a, 8, 9, 11) | E | E | G | M | O | R | R | S | A | E | T | X | M | P | L | E |
| merge(a, 12, 13, 15) | E | E | G | M | O | R | R | S | A | E | T | X | E | L | M | P |
| merge(a, 0, 3, 7) | E | E | G | M | O | R | R | S | T | E | X | A | M | P | L | E |
| merge(a, 8, 11, 15) | E | E | G | M | O | R | R | S | A | E | E | L | M | P | T | X |
| merge(a, 0, 7, 15) | A | E | E | E | E | G | L | M | M | O | P | R | R | S | T | X |

**Trace of merge results for bottom-up mergesort**

**A.** Yes, if merge is stable.



## Sorting challenge 5D (continued)

**Q.** Is merge stable?

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

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



## 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").

## 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++];

}
```

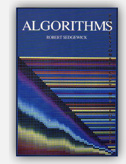
| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | lo | | | i | mid | | | j | | hi |
| aux[] | A | G | L | O | R | H | I | M | S | T |
| | | | | | | | k | | | |
| a[] | A | G | H | I | L | M | | | | |



## Postscript: optimizing mergesort (a short history)

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

```
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++];
```




| | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 0 | | | i | | M | | | | j | | | N |
| a[] | A | G | L | O | R | ∞ | b[] | H | I | M | S | T | ∞ |

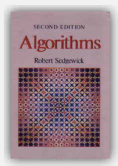
| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | k | | | |
| aux[] | A | G | H | I | L | M | | | | |

**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.




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

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

   int i = lo, j = hi;
   for (int k = lo; k <= hi; k++)                          merge
      if (less(aux[j], aux[i])) a[k] = aux[j--];
      else                      a[k] = aux[i++];
}
```

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | lo | | | i | mid | | j | | | hi |
| aux[] | A | G | L | O | R | T | S | M | I | H |
| | | | | | | | k | | | |
| a[] | A | G | H | I | L | M | | | | |

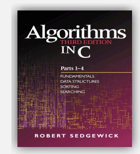
**Problem.** Copy still in inner loop.



## Postscript: Optimizing mergesort (a short history)

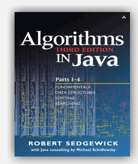
**Idea 3 (1990s).** Eliminate copy with recursive argument switch.

```
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);
```




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


`Arrays.sort()`



## Sorting challenge 6

Problem. Choose mergesort for Algs 4th edition.
Recursive argument switch is out (recommended only for pros).

Q. Why not use reverse array copy?

```
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++];
}
```