

INTRODUCTION TO DATA STRUCTURES and ALGORITHMS

Rutgers University

MERGESORT

- *mergesort*
- *bottom-up mergesort*
- *sorting complexity*
- *divide-and-conquer*

<http://algs4.cs.princeton.edu>



<http://ds.cs.rutgers.edu>

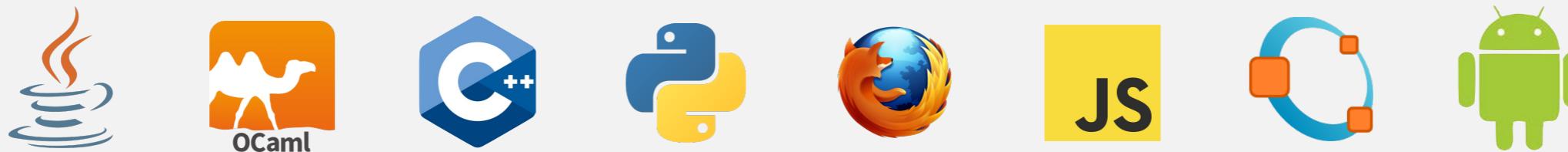
Some slides Adopted and modified from Sedgewick and Wayne

Two classic sorting algorithms: mergesort and quicksort

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. [this lecture]



Quicksort. [next lecture]



MERGESORT

- *mergesort*
- *bottom-up mergesort*
- *sorting complexity*
- *divide-and-conquer*

Mergesort

Basic plan.

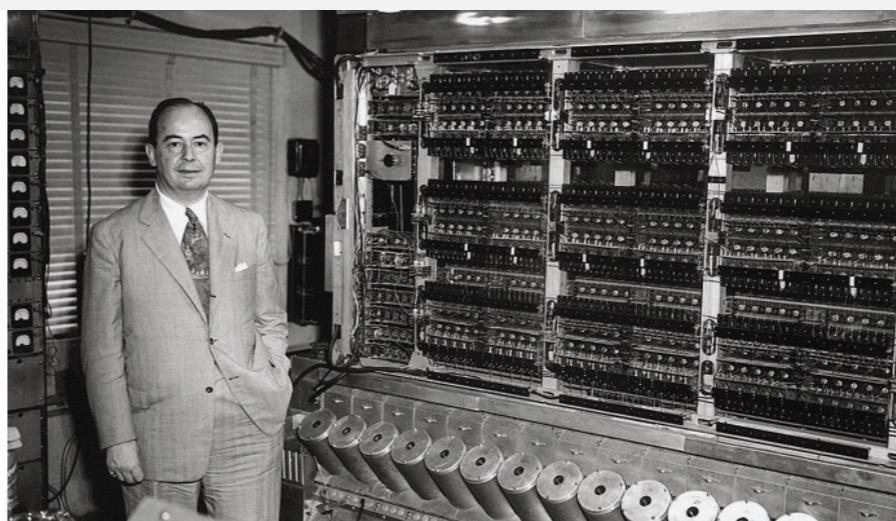
- Divide array into two halves.
- Recursively sort each half.
- Merge two halves.

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

Mergesort overview

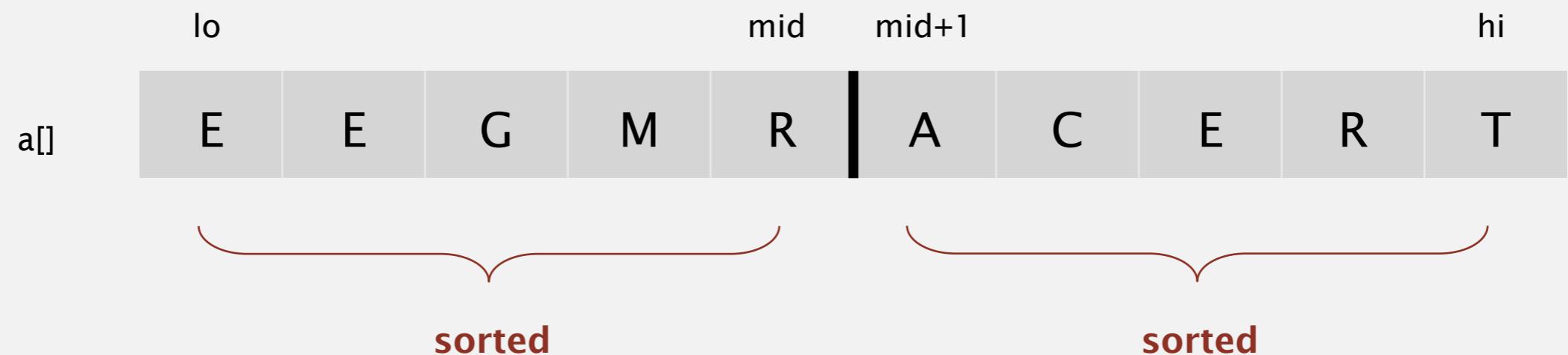
First Draft of a Report on the EDVAC

John von Neumann

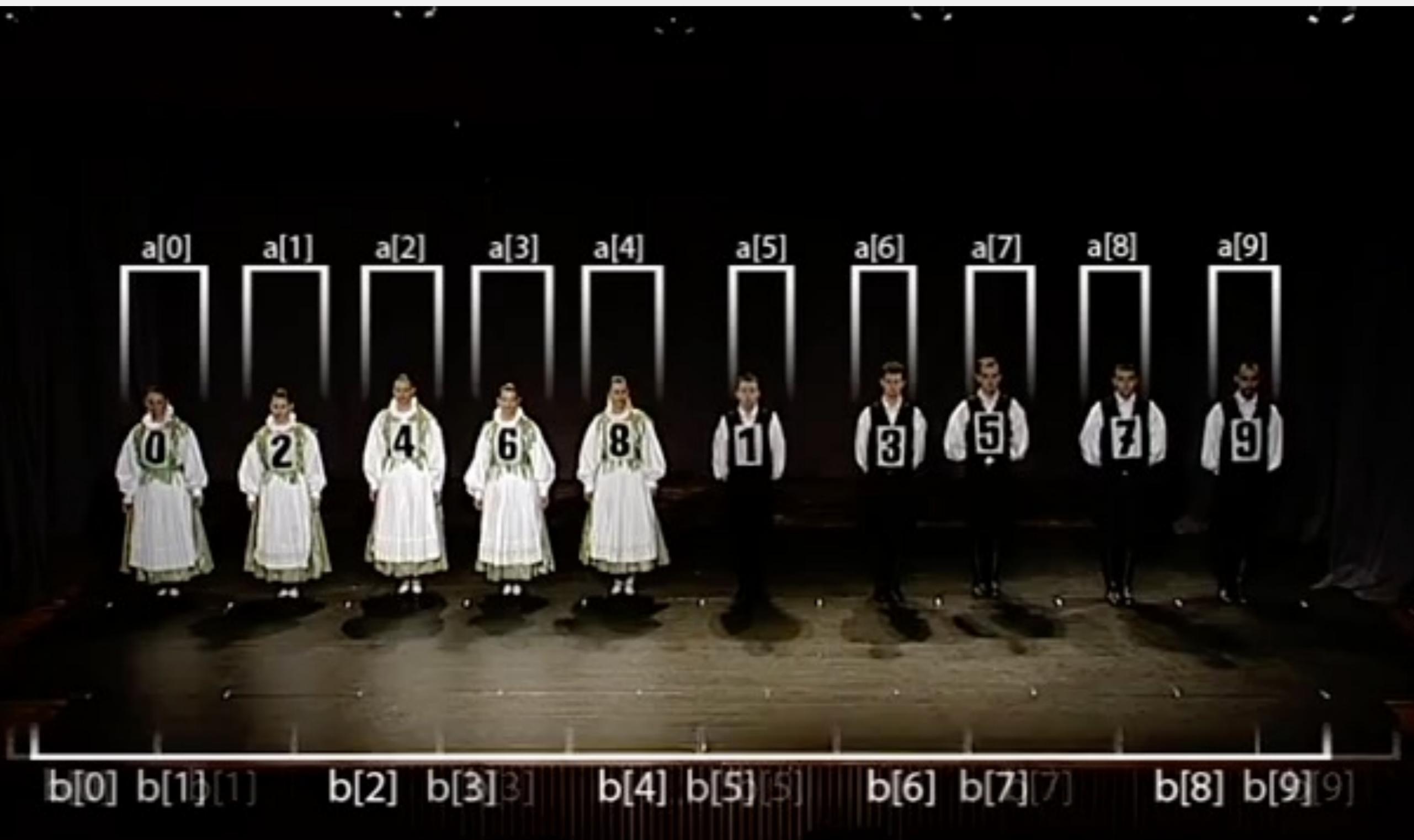


Abstract in-place merge demo

Goal. Given two sorted subarrays $a[lo]$ to $a[mid]$ and $a[mid+1]$ to $a[hi]$, replace with sorted subarray $a[lo]$ to $a[hi]$.



Mergesort: Transylvanian–Saxon folk dance



http://www.youtube.com/watch?v=XaqR3G_NVoo

Merging: Java implementation

```
private static void merge(Comparable[] a, Comparable[] aux, int lo, int mid, int hi)
```

```
{
```

```
    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++)
```

merge

```
{
```

```
        if (i > mid) a[k] = aux[j++];
```

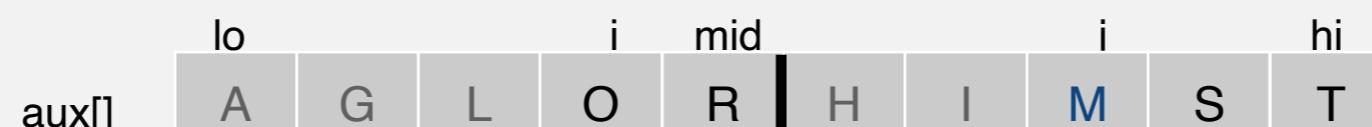
```
        else if (j > hi) a[k] = aux[i++];
```

```
        else if (less(aux[j], aux[i])) a[k] = aux[j++];
```

```
        else a[k] = aux[i++];
```

```
}
```

```
}
```



Mergesort quiz 1

How many calls does merge() make to less() in order to merge two sorted subarrays, each of length $n / 2$, into a sorted array of length n ?

- A. $\sim \frac{1}{4} n$ to $\sim \frac{1}{2} n$
- B. $\sim \frac{1}{2} n$
- C. $\sim \frac{1}{2} n$ to $\sim n$
- D. $\sim n$

Mergesort: Java implementation

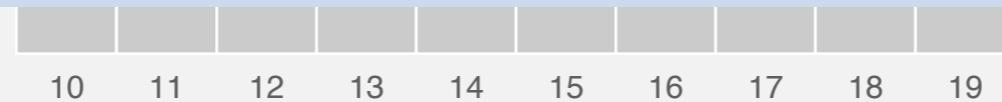
```
public class Merge
{
    private static void merge(...)

    { /* 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)
{  
    Comparable[] aux = new Comparable[a.length];  
    sort(a, aux, 0, a.length - 1);  
    mid  
}  
}
```



Mergesort: trace

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

result after recursive call

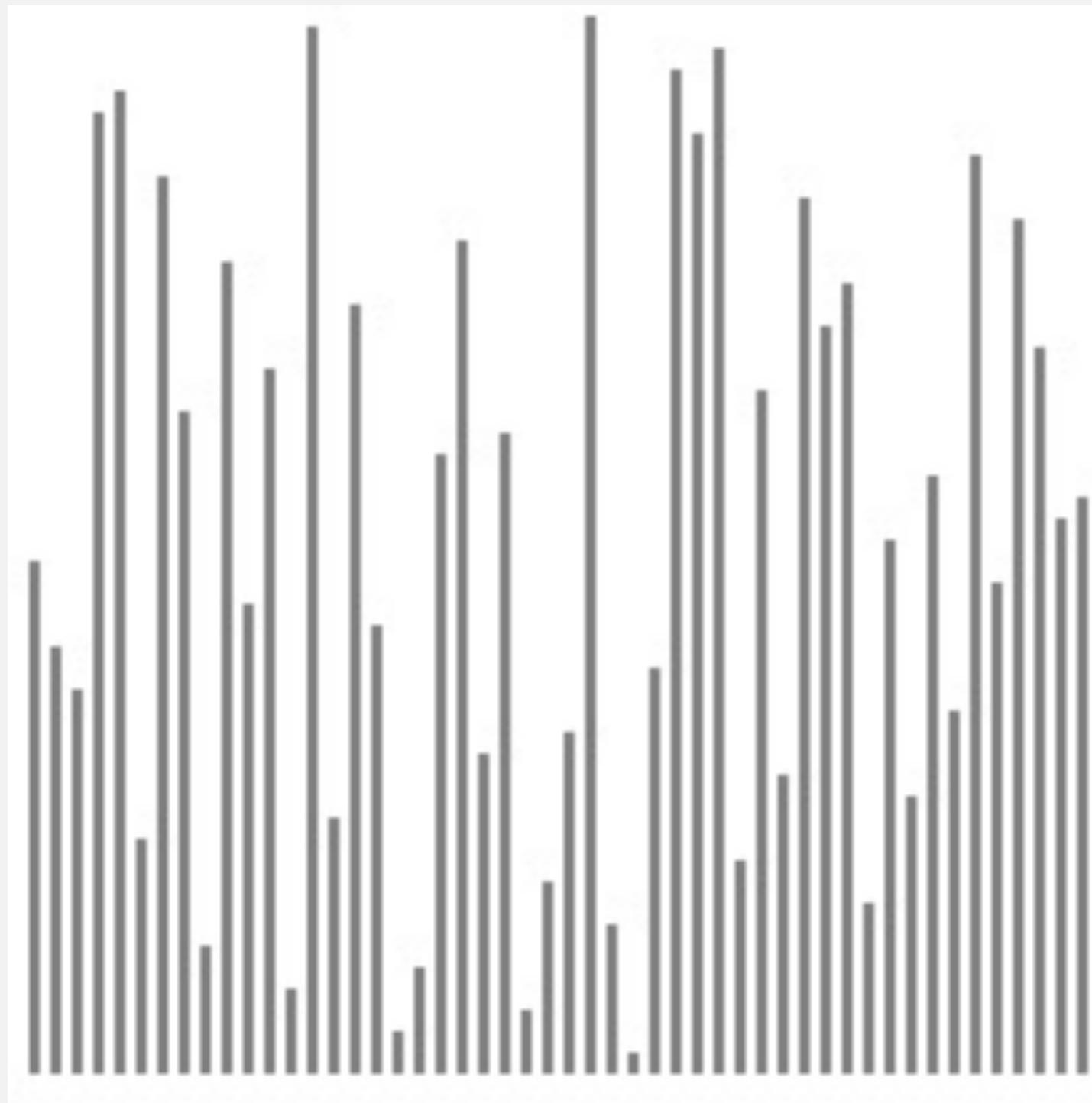
Mergesort quiz 2

Which of the following subarray lengths will occur when running mergesort on an array of length 12?

- A. { 1, 2, 3, 4, 6, 8, 12 }
- B.** { 1, 2, 3, 6, 12 }
- C. { 1, 2, 4, 8, 12 }
- D. { 1, 3, 6, 9, 12 }

Mergesort: animation

50 random items

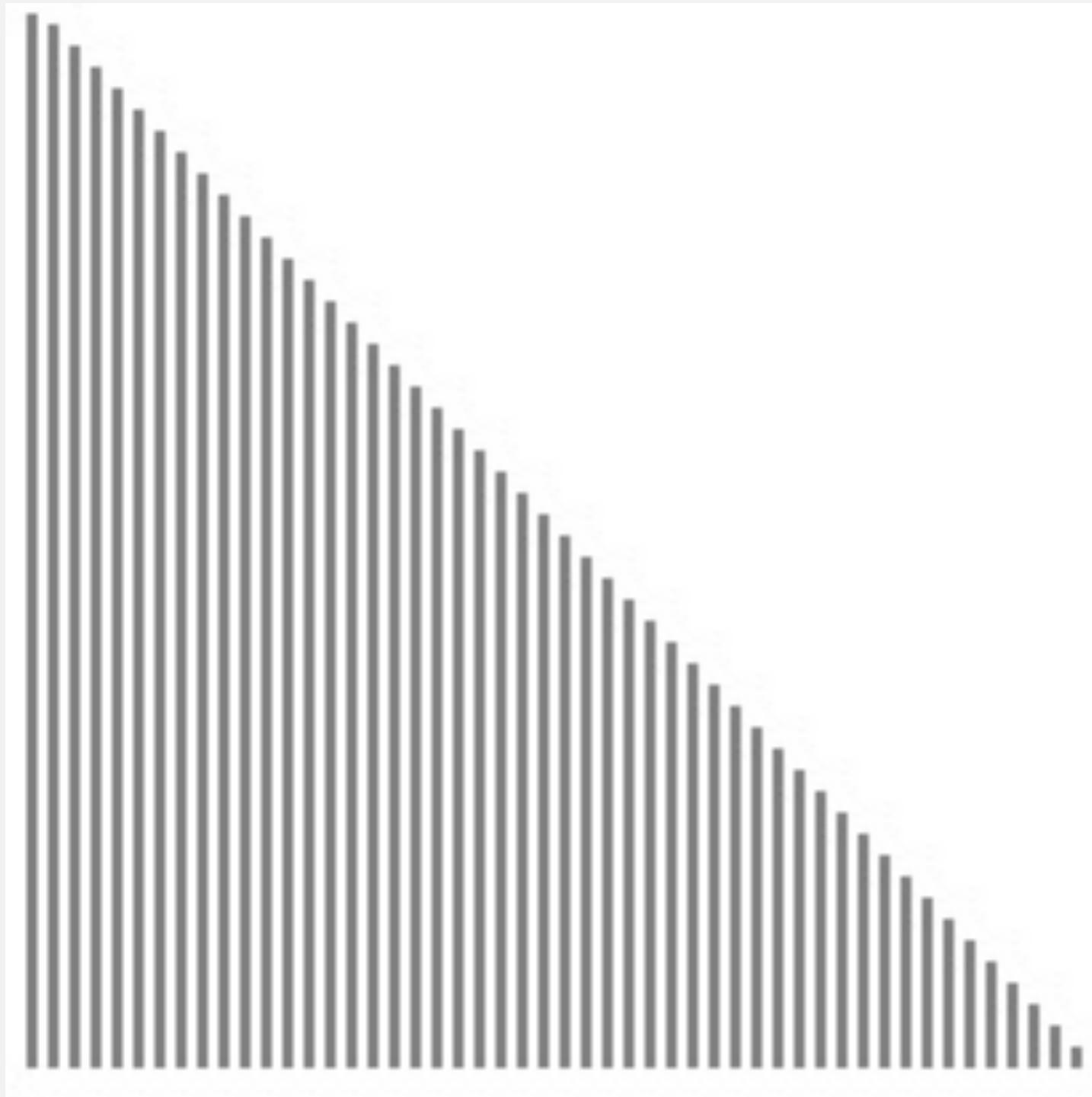


- ▲ algorithm position
- in order
- current subarray
- not in order

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

Mergesort: animation

50 reverse-sorted items



- ▲ algorithm position
- in order
- current subarray
- not in order

<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.

	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 analysis: number of compares

Proposition. Mergesort uses $\leq n \lg n$ compares to sort an array of length n .

Pf sketch. The number of compares $C(n)$ to mergesort an array of length n satisfies the recurrence:

$$C(n) \leq C(\lceil n/2 \rceil) + C(\lfloor n/2 \rfloor) + n - 1 \quad \text{for } n > 1, \text{ with } C(1) = 0.$$



left half right half merge

We solve this simpler recurrence, and assume n is a power of 2:

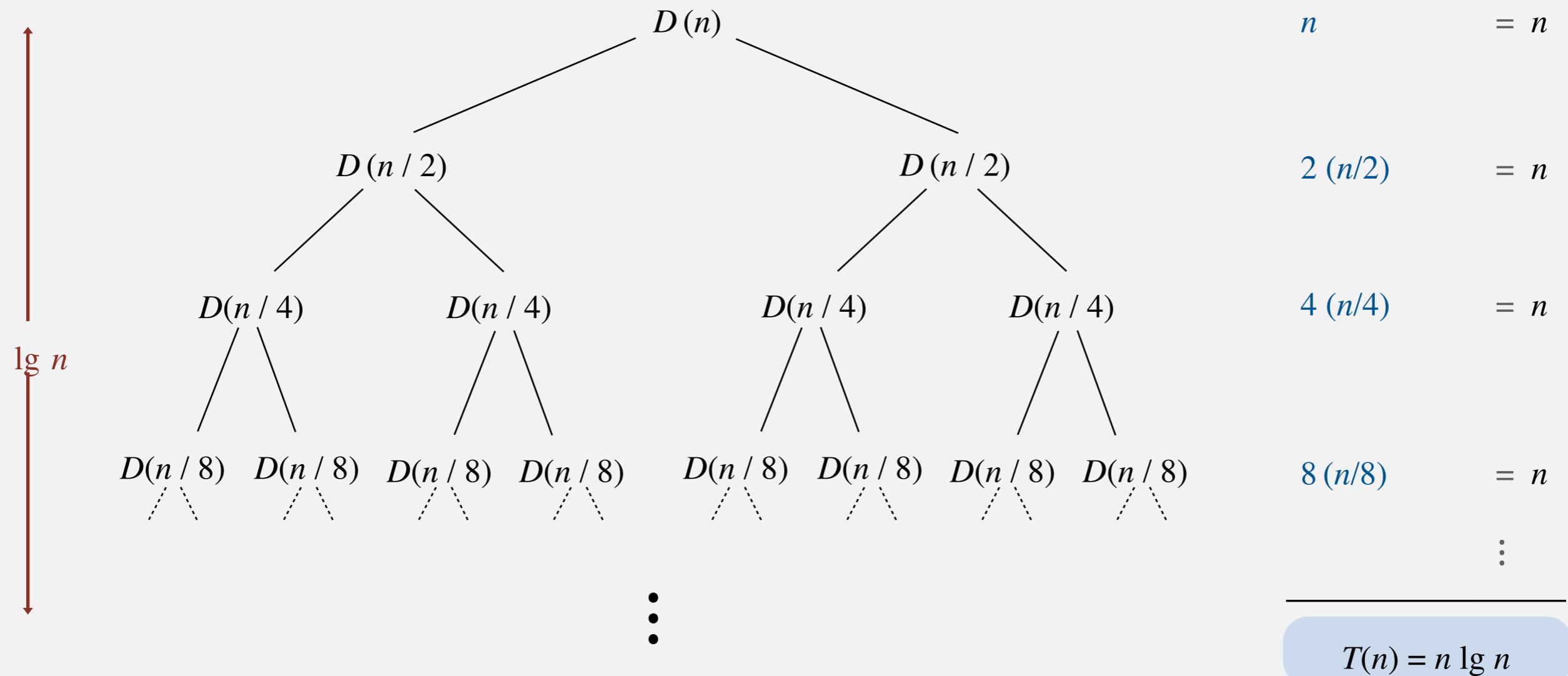
$$D(n) = 2D(n/2) + n, \text{ for } n > 1, \text{ with } D(1) = 0.$$


result holds for all n
(analysis cleaner in this case)

Divide-and-conquer recurrence

Proposition. If $D(n)$ satisfies $D(n) = 2 D(n / 2) + n$ for $n > 1$, with $D(1) = 0$, then $D(n) = n \lg n$.

Pf by picture. [assuming n is a power of 2]



Mergesort analysis: number of array accesses

Proposition. Mergesort uses $\leq 6n \lg n$ array accesses to sort an array of length n .

Pf sketch. The number of array accesses $A(n)$ satisfies the recurrence:

$$A(n) \leq A(\lceil n/2 \rceil) + A(\lfloor n/2 \rfloor) + 6n \text{ for } n > 1, \text{ with } A(1) = 0.$$

Key point. Any algorithm with the following structure takes $n \log n$ time:

```
public static void f(int n)
{
    if (n == 0) return;
    f(n/2);           ← solve two problems
    f(n/2);           ← of half the size
    linear(n);        ← do a linear amount of work
}
```

Notable examples. FFT, hidden-line removal, Kendall-tau distance, ...

Mergesort analysis: memory

Proposition. Mergesort uses extra space proportional to n .

Pf. The array `aux[]` needs to be of length n for the last merge.

two sorted subarrays

A C D G H I M N U V

B E F J O P Q R S T

merged result

A B C D E F G H I J M N O P Q R S T U V

Def. A sorting algorithm is **in-place** if it uses $\leq c \log n$ extra memory.

Ex. Insertion sort and selection sort.

Challenge 1 (not hard). Use `aux[]` array of length $\sim \frac{1}{2} n$ instead of n .

Challenge 2 (very hard). In-place merge. [Kronrod 1969]

Mergesort quiz 3

Is our implementation of mergesort **stable**?

- A. Yes.
- B. No, but it can be easily modified to be stable.
- C. No, mergesort is inherently unstable.
- D. *I don't remember what stability means.*



a sorting algorithm is stable if it
preserves the relative order of equal keys

input	C	A ₁	B	A ₂	A ₃
sorted	A ₃	A ₁	A ₂	B	C

not stable

Stability: mergesort

Proposition. Mergesort is **stable**.

```
public class Merge
{
    private static void merge(...)
    { /* as before */ }

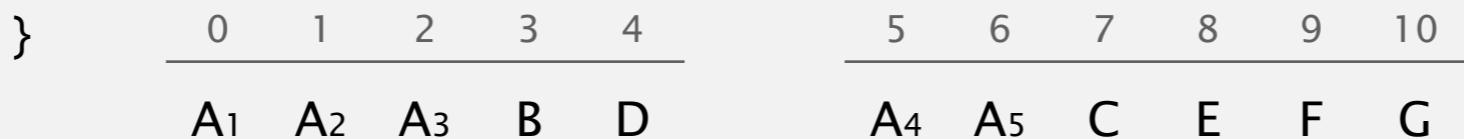
    private static void sort(Comparable[] a, Comparable[] aux, int lo, int hi)
    {
        if (hi <= lo) return;
        int mid = lo + (hi - lo) / 2;
        sort(a, aux, lo, mid);
        sort(a, aux, mid+1, hi);
        merge(a, aux, lo, mid, hi);
    }
}
```

Pf. { /* as before */ } Suffices to verify that merge operation is stable.

Stability: mergesort

Proposition. Merge operation is **stable**.

```
private static void merge(...)  
{  
    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++];  
    }  
}
```



Pf. Takes from left subarray if equal keys.

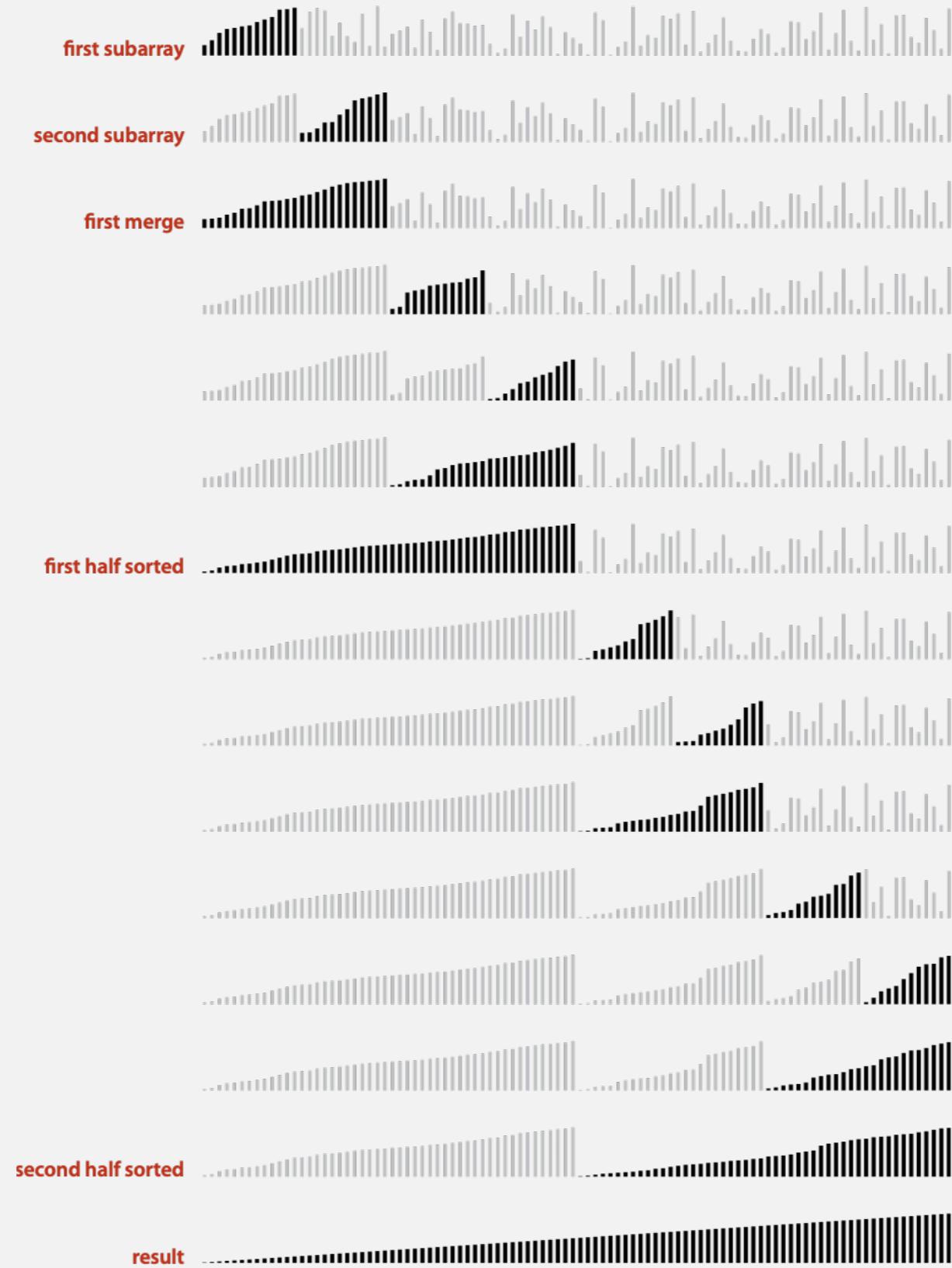
Mergesort: practical improvements

Use insertion sort for small subarrays.

- Mergesort has too much overhead for tiny subarrays.
- Cutoff to insertion sort for ≈ 10 items.

```
private static void sort(...)  
{  
    if (hi <= lo + CUTOFF - 1)  
    {  
        Insertion.sort(a, lo, hi);  
        return;  
    }  
  
    int mid = lo + (hi - lo) / 2;  
    sort (a, aux, lo, mid);  
    sort (a, aux, mid+1, hi);  
    merge(a, aux, lo, mid, hi);  
}
```

Mergesort with cutoff to insertion sort: visualization



Mergesort: practical improvements

Stop if already sorted.

- Is largest item in first half \leq smallest item in second half?
- Helps for partially ordered arrays.



A B C D E F G H I J M N O P Q R S T U V

```
private static void sort(...)  
{  
    if (hi <= lo) return;  
    int mid = lo + (hi - lo) / 2;  
    sort (a, aux, lo, mid);  
    sort (a, aux, mid+1, hi);  
    if (!less(a[mid+1], a[mid])) return;  
    merge(a, aux, lo, mid, hi);  
}
```

Mergesort: practical improvements

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.

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

private static void sort(Comparable[] a, Comparable[] aux, int lo, int hi)
{
    if (hi <= lo) return;
    int mid = lo + (hi - lo) / 2;
    sort (aux, a, lo, mid);
    sort (aux, a, mid+1, hi);
    merge(a, aux, lo, mid, hi);
}
```

← merge from a[] to aux[]

↑
assumes aux[] is initialize to a[] once,
before recursive calls

switch roles of aux[] and a[]

Java 6 system sort

Basic algorithm in `Arrays.sort()` for sorting objects = mergesort.

- Cutoff to insertion sort = 7.
- Stop-if-already-sorted test.
- Eliminate-the-copy-to-the-auxiliary-array trick.



<http://hg.openjdk.java.net/jdk6/jdk6/jdk/file/tip/src/share/classes/java/util/Arrays.java>

MERGESORT

- *mergesort*
- *bottom-up mergesort*
- *sorting complexity*
- *divide-and-conquer*

Bottom-up mergesort

Basic plan.

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

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

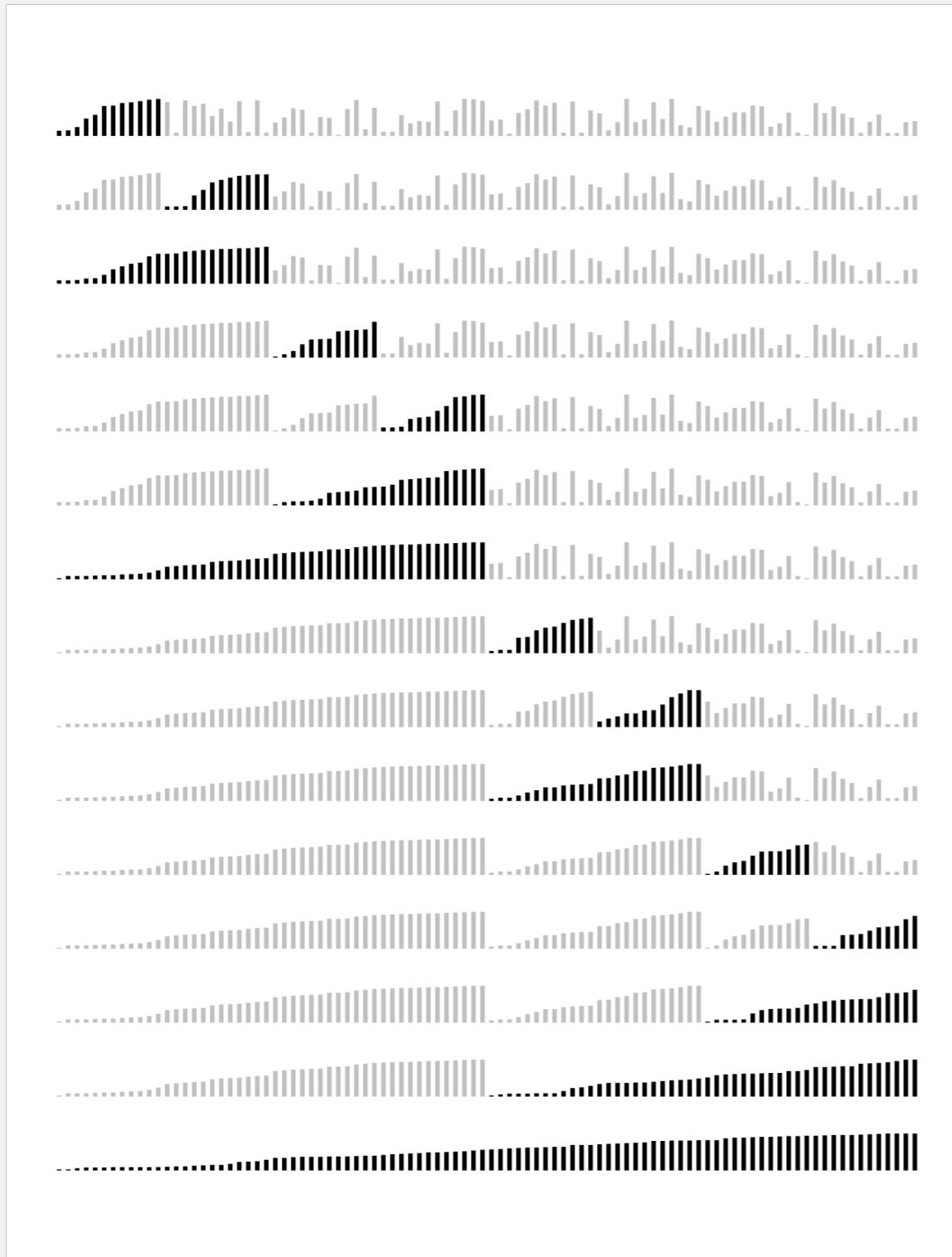
Bottom-up mergesort: Java implementation

```
public class MergeBU
{
    private static void merge(...)
    { /* as before */

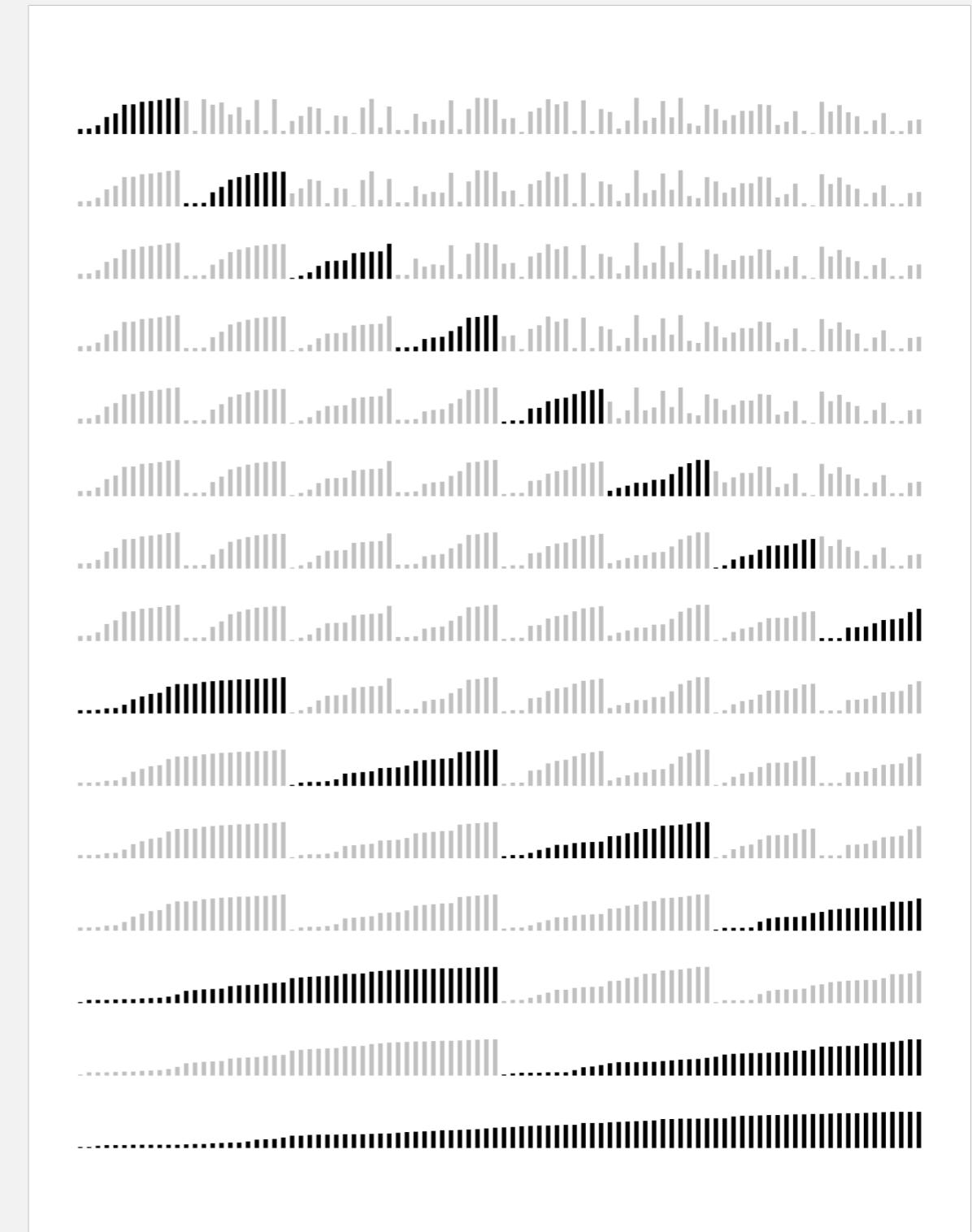
    public static void sort(Comparable[] a)
    {
        int n = a.length;
        Comparable[] aux = new Comparable[n];
        for (int sz = 1; sz < n; sz = sz+sz)
            for (int lo = 0; lo < n-sz; lo += sz+sz)
                merge(a, aux, lo, lo+sz-1, Math.min(lo+sz+sz-1, n-1));
    }
}
```

Bottom line. Simple and non-recursive version of mergesort.

Mergesort: visualizations



top-down mergesort (cutoff = 12)



bottom-up mergesort (cutoff = 12)

Mergesort quiz 4

Which is faster in practice: top-down mergesort or bottom-up mergesort? You may assume n is a power of 2.

- A. Top-down (recursive) mergesort.
- B. Bottom-up (non-recursive) mergesort.
- C. No observable difference.
- D. *I don't know.*

Natural mergesort

Idea. Exploit pre-existing order by identifying naturally occurring runs.

input

1	5	10	16	3	4	23	9	13	2	7	8	12	14
---	---	----	----	---	---	----	---	----	---	---	---	----	----

first run

1	5	10	16	3	4	23	9	13	2	7	8	12	14
---	---	----	----	---	---	----	---	----	---	---	---	----	----

second run

1	5	10	16	3	4	23	9	13	2	7	8	12	14
---	---	----	----	---	---	----	---	----	---	---	---	----	----

merge two runs

1	3	4	5	10	16	23	9	13	2	7	8	12	14
---	---	---	---	----	----	----	---	----	---	---	---	----	----

Tradeoff. Fewer passes vs. extra compares per pass to identify runs.

Timsort

- Natural mergesort.
- Use binary insertion sort to make initial runs (if needed).
- A few more clever optimizations.

Intro

This describes an adaptive, stable, natural mergesort, modestly called timsort (hey, I earned it <wink>). It has supernatural performance on many kinds of partially ordered arrays (less than $\lg(n!)$ comparisons needed, and as few as $n-1$), yet as fast as Python's previous highly tuned samplesort hybrid on random arrays.



Tim Peters

In a nutshell, the main routine marches over the array once, left to right, alternately identifying the next run, then merging it into the previous runs "intelligently". Everything else is complication for speed, and some hard-won measure of memory efficiency.

Consequence. Linear time on many arrays with pre-existing order.

Now widely used. Python, Java 7–9, GNU Octave, Android,

<http://hg.openjdk.java.net/jdk7/jdk7/jdk/file/tip/src/share/classes/java/util/Arrays.java>

[Envisage](#)[About Envisage](#)[Follow Envisage](#)[Dissemination](#)[Log in](#)

Proving that Android's, Java's and Python's sorting algorithm is broken (and showing how to fix it)

⌚ February 24, 2015

📁 Envisage

Written by Stijn de Gouw. 🚀 \$s

Tim Peters developed the [Timsort hybrid sorting algorithm](#) in 2002. It is a clever combination of ideas from merge sort and insertion sort, and designed to perform well on real world data. TimSort was first developed for Python, but later ported to Java (where it appears as `java.util.Collections.sort` and `java.util.Arrays.sort`) by [Joshua Bloch](#) (the designer of Java Collections who also pointed out that [most binary search algorithms were broken](#)). TimSort is today used as the default sorting algorithm for Android SDK, Sun's JDK and OpenJDK. Given the popularity of these platforms this means that the number of computers, cloud services and mobile phones that use TimSort for sorting is well into the billions.

<http://envisage-project.eu/proving-android-java-and-python-sorting-algorithm-is-broken-and-how-to-fix-it>

Sorting summary

	inplace?	stable?	best	average	worst	remarks
selection	✓		$\frac{1}{2} n^2$	$\frac{1}{2} n^2$	$\frac{1}{2} n^2$	n exchanges
insertion	✓	✓	n	$\frac{1}{4} n^2$	$\frac{1}{2} n^2$	use for small n or partially ordered
shell	✓		$n \log_3 n$?	$c n^{3/2}$	tight code; subquadratic
merge		✓	$\frac{1}{2} n \lg n$	$n \lg n$	$n \lg n$	$n \log n$ guarantee; stable
timsort		✓	n	$n \lg n$	$n \lg n$	improves mergesort when pre-existing order
?	✓	✓	n	$n \lg n$	$n \lg n$	holy sorting grail

MERGESORT

- ▶ *mergesort*
- ▶ *bottom-up mergesort*
- ▶ *sorting complexity*
- ▶ *divide-and-conquer*

Complexity of sorting

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

Model of computation. Allowable operations.

Cost model. Operation counts.

Upper bound. Cost guarantee provided by **some** algorithm for X .

Lower bound. Proven limit on cost guarantee of **all** algorithms for X .

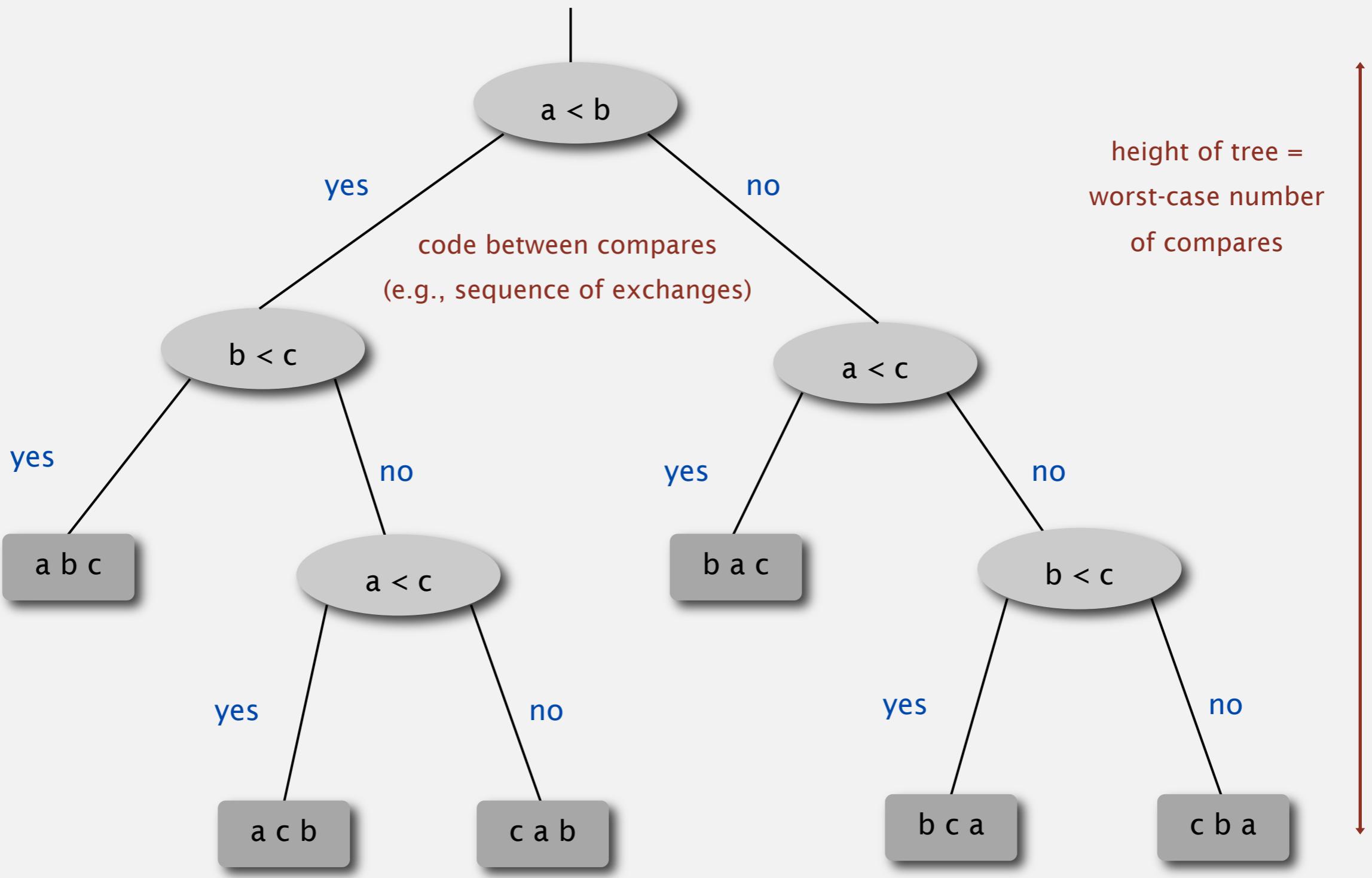
Optimal algorithm. Algorithm with best possible cost guarantee for X .



lower bound \sim upper bound

model of computation	<i>decision tree</i>	←	can access information only through compares (e.g., Java Comparable framework)
cost model	# compares		
upper bound	$\sim n \lg n$ from mergesort		
lower bound	?		
optimal algorithm	?		

Decision tree (for 3 distinct keys a, b, and c)



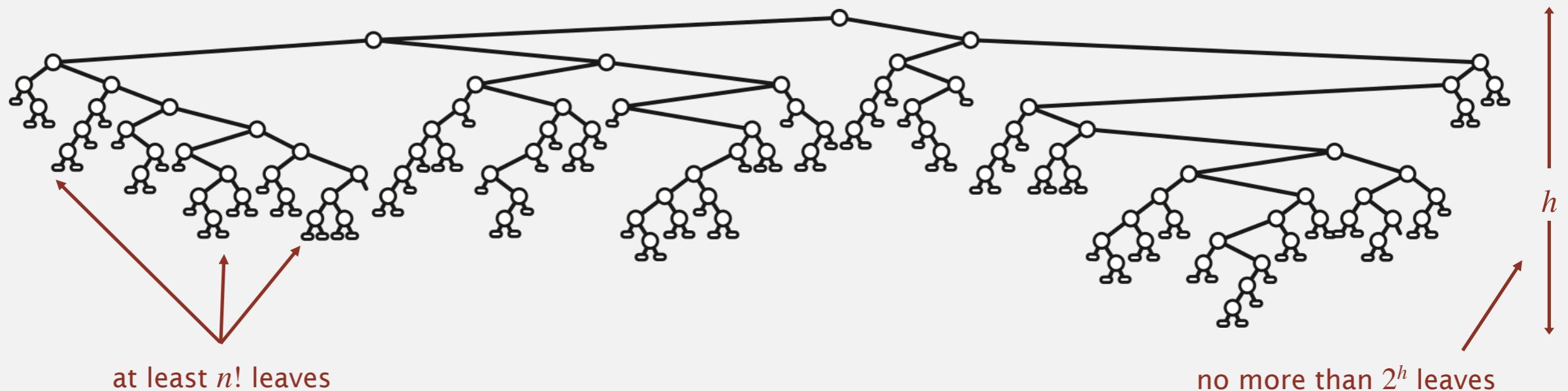
each leaf corresponds to one (and only one) ordering;
(at least) one leaf for each possible ordering

Compare-based lower bound for sorting

Proposition. Any compare-based sorting algorithm must make at least $\lg(n !) \sim n \lg n$ 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.



Compare-based lower bound for sorting

Proposition. Any compare-based sorting algorithm must make at least $\lg(n !) \sim n \lg n$ 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.

$$\begin{aligned} 2^h &\geq \# \text{leaves} \geq n ! \\ \Rightarrow h &\geq \lg(n !) \sim n \lg n \end{aligned}$$

↑
Stirling's formula

Complexity of sorting

Model of computation. Allowable operations.

Cost model. Operation count(s).

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 possible cost guarantee for X .

model of computation	<i>decision tree</i>
cost model	# compares
upper bound	$\sim n \lg n$
lower bound	$\sim n \lg n$
optimal algorithm	<i>mergesort</i>

complexity of sorting

First goal of algorithm design: optimal algorithms.

Complexity results in context

Compares? Mergesort **is** optimal with respect to number compares.

Space? Mergesort **is not** optimal with respect to space usage.



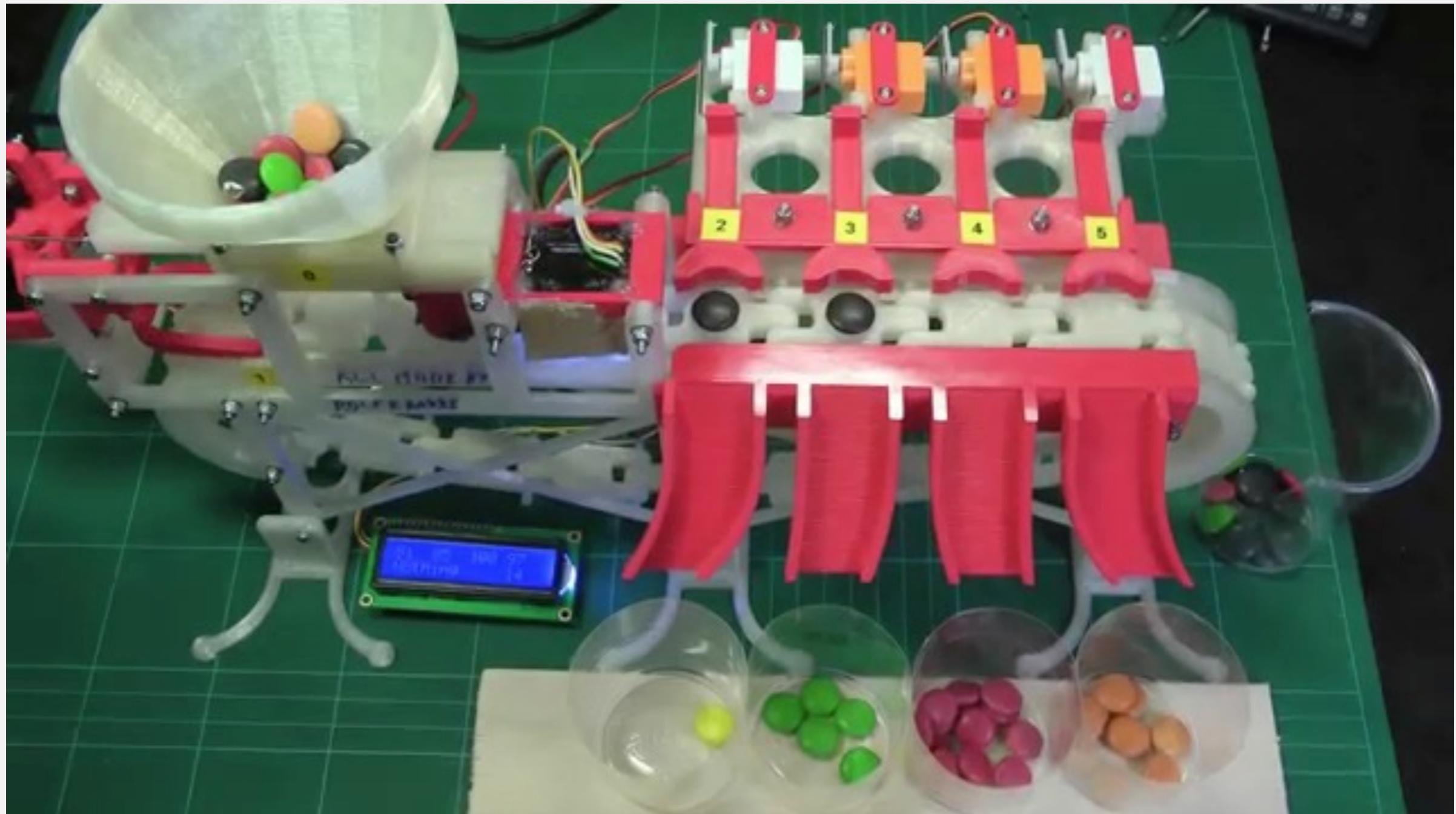
Lessons. Use theory as a guide.

Ex. Design sorting algorithm that guarantees $\sim \frac{1}{2} n \lg n$ compares?

Ex. Design sorting algorithm that is both time- and space-optimal?

Commercial break

Q. Why doesn't this Skittles machine violate the sorting lower bound?



<https://www.youtube.com/watch?v=tSEHDBSynVo>

Complexity results in context (continued)

Lower bound may not hold if the algorithm can take advantage of:

- The initial order of the input array.
Ex: insertion sort requires only a linear number of compares on partially sorted arrays.
- The distribution of key values.
Ex: 3-way quicksort requires only a linear number of compares on arrays with a constant number of distinct keys. [stay tuned]
- The representation of the keys.
Ex: radix sorts require no key compares — they access the data via character/digit compares. [stay tuned]

COMMONLY USED NOTATIONS IN THE THEORY OF ALGORITHMS

notation	provides	example	shorthand for
Tilde	leading term	$\sim \frac{1}{2} n^2$	$\frac{1}{2} n^2$ $\frac{1}{2} n^2 + 22 n \log n + 3 n$
Big Theta	order of growth	$\Theta(n^2)$	$\frac{1}{2} n^2$ $10 n^2$ $5 n^2 + 22 n \log n + 3 n$
Big O	upper bound	$O(n^2)$	$10 n^2$ $100 n$ $22 n \log n + 3 n$
Big Omega	lower bound	$\Omega(n^2)$	$\frac{1}{2} n^2$ n^5 $n^3 + 22 n \log n + 3 n$

MERGESORT

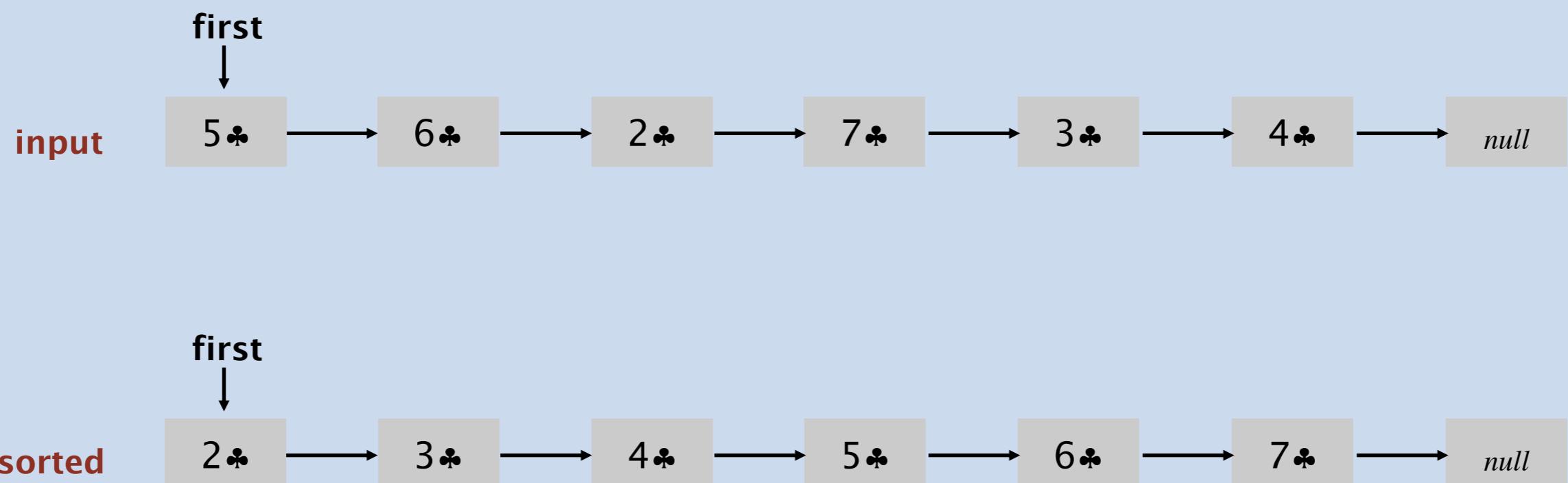
- *mergesort*
- *bottom-up mergesort*
- *sorting complexity*
- ***divide-and-conquer***

SORTING A LINKED LIST

Problem. Given a singly linked list, rearrange its nodes in sorted order.

Version 1. Linearithmic time, linear extra space.

Version 2. Linearithmic time, logarithmic (or constant) extra space.



INTRODUCTION TO DATA STRUCTURES and ALGORITHMS

Rutgers University

MERGE SORT

-
- ▶ *mergesort*
 - ▶ *bottom-up mergesort*
 - ▶ *sorting complexity*
 - ▶ *divide-and-conquer*

