https://algs4.cs.princeton.edu

# 2.2 MERGESORT

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

Last updated on 2/19/19 8:24 AM

# 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]

…

https://algs4.cs.princeton.edu

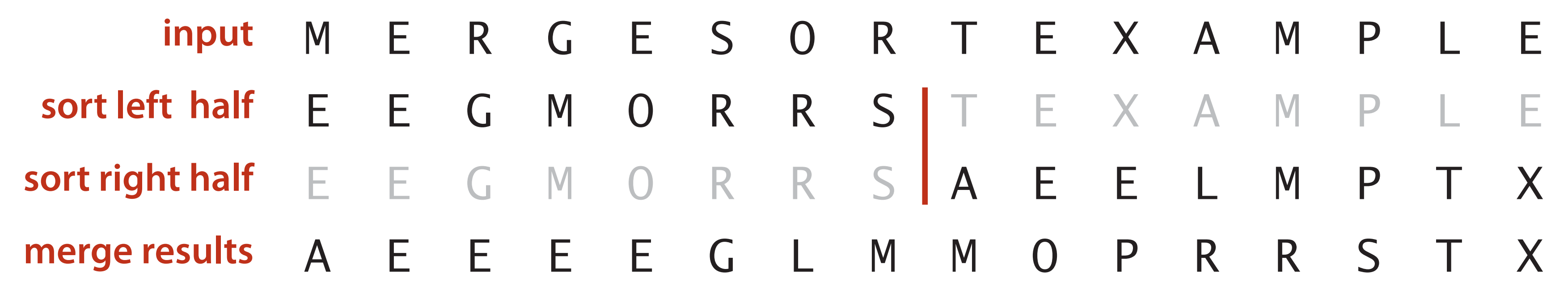
# 2.2 MERGESORT

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

# Mergesort

Basic plan.

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

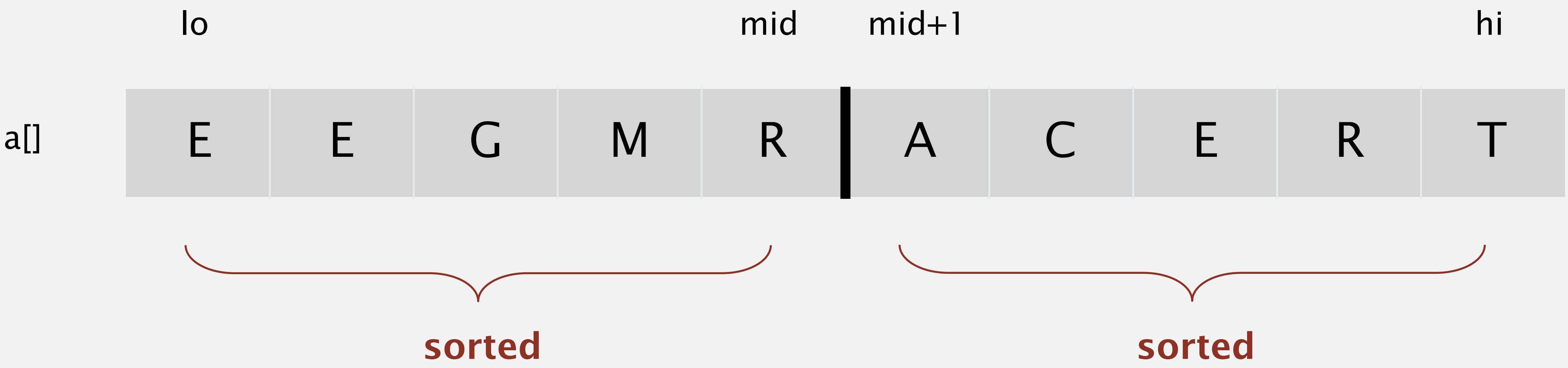


| | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| input | M | E | R | G | E | S | O | R | T | E | X | A | M | P | L | E |
| sort left half | E | E | G | M | O | R | R | S | T | E | X | A | M | P | L | E |
| sort right half | E | E | G | M | O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

# 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]`.



# Merging: Java implementation

```
private static void merge(Comparable[] a, Comparable[] aux, 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++];
   }

}
```

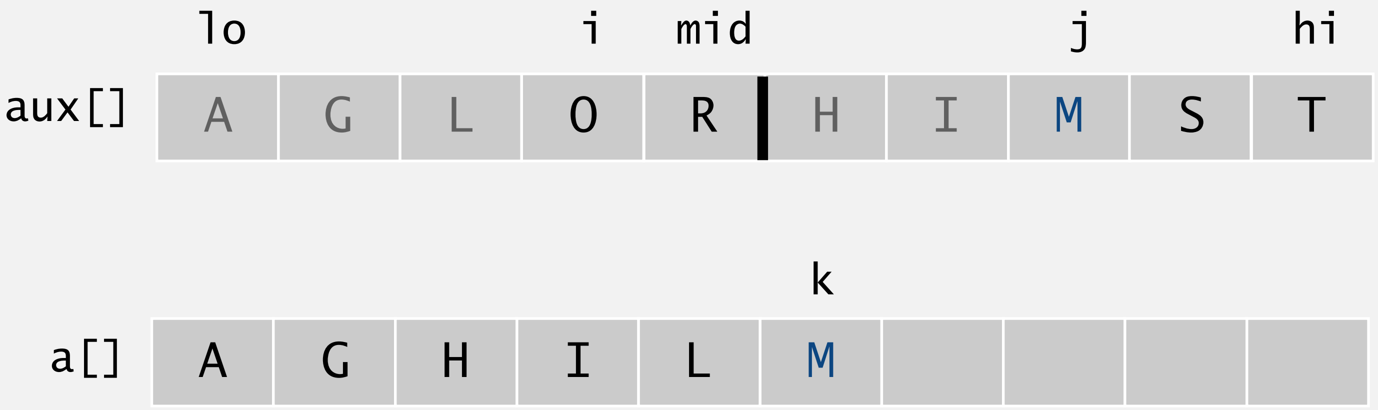
**copy**

**merge**

Bad programming practice.
Used here to save vertical space.

Equivalent to:

```
{
   a[k] = aux[j];
   j++;
}
```



# 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 $\sim \frac{1}{2} n$

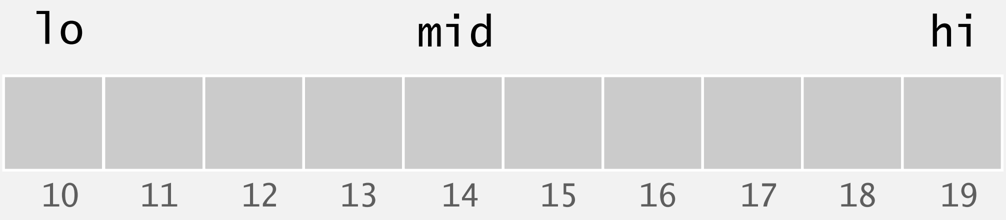
**B.** $\sim \frac{1}{2} n$

**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);
   }
}
```



# Mergesort: trace

lo hi

a[]

| | 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, 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, 0, 1, 3)` | **E** | **G** | **M** | **R** | E | S | O | R | T | E | X | A | M | P | L | E |
| `merge(a, aux, 4, 4, 5)` | E | G | M | R | **E** | **S** | O | R | T | E | X | A | M | P | L | E |
| `merge(a, aux, 6, 6, 7)` | E | G | M | R | E | S | **O** | **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** | **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.** $\{ 1, 2, 3, 4, 6, 8, 12 \}$

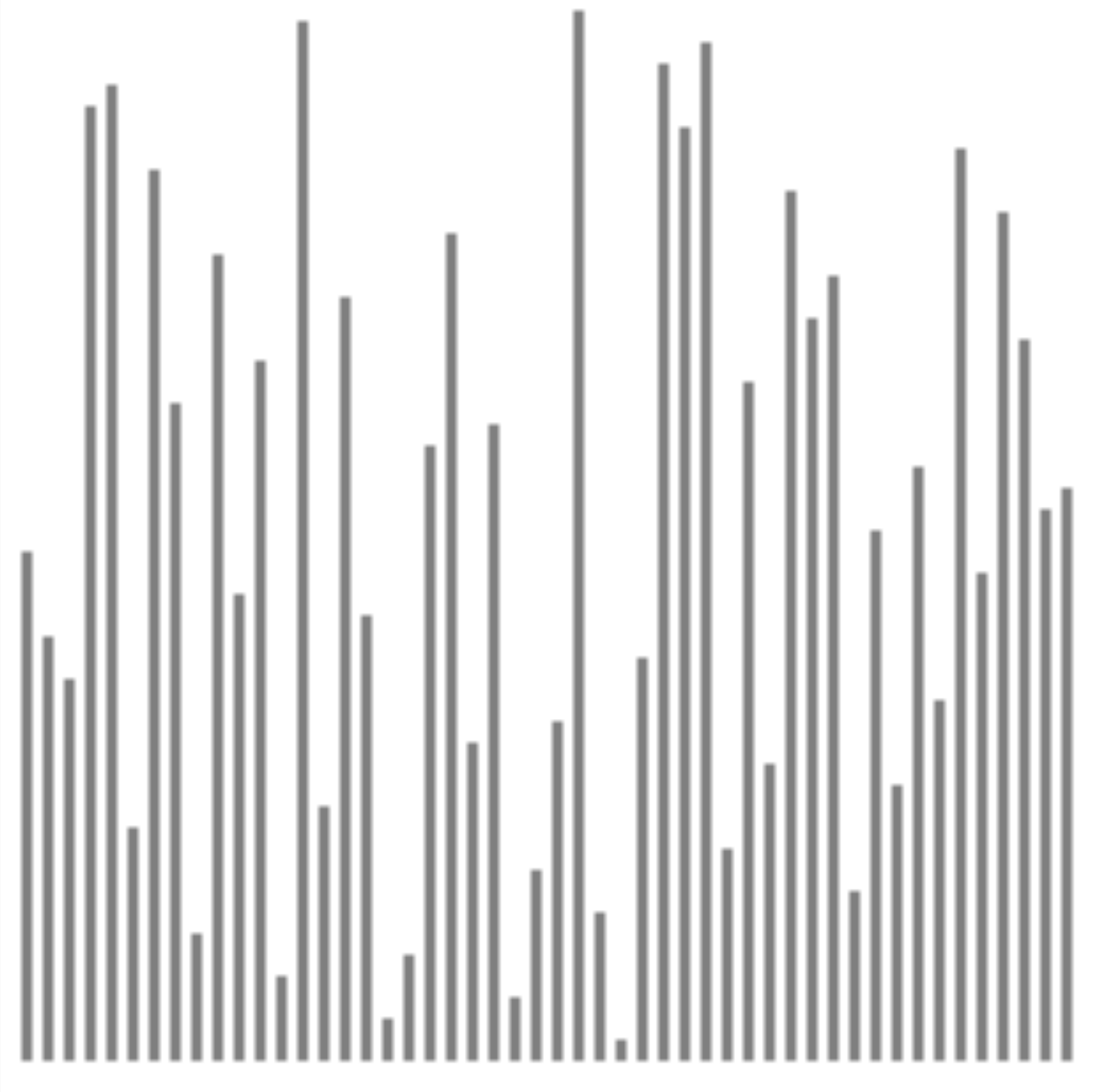
**B.** $\{ 1, 2, 3, 6, 12 \}$

**C.** $\{ 1, 2, 4, 8, 12 \}$

**D.** $\{ 1, 3, 6, 9, 12 \}$

# Mergesort: animation

50 random items



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

▲ algorithm position

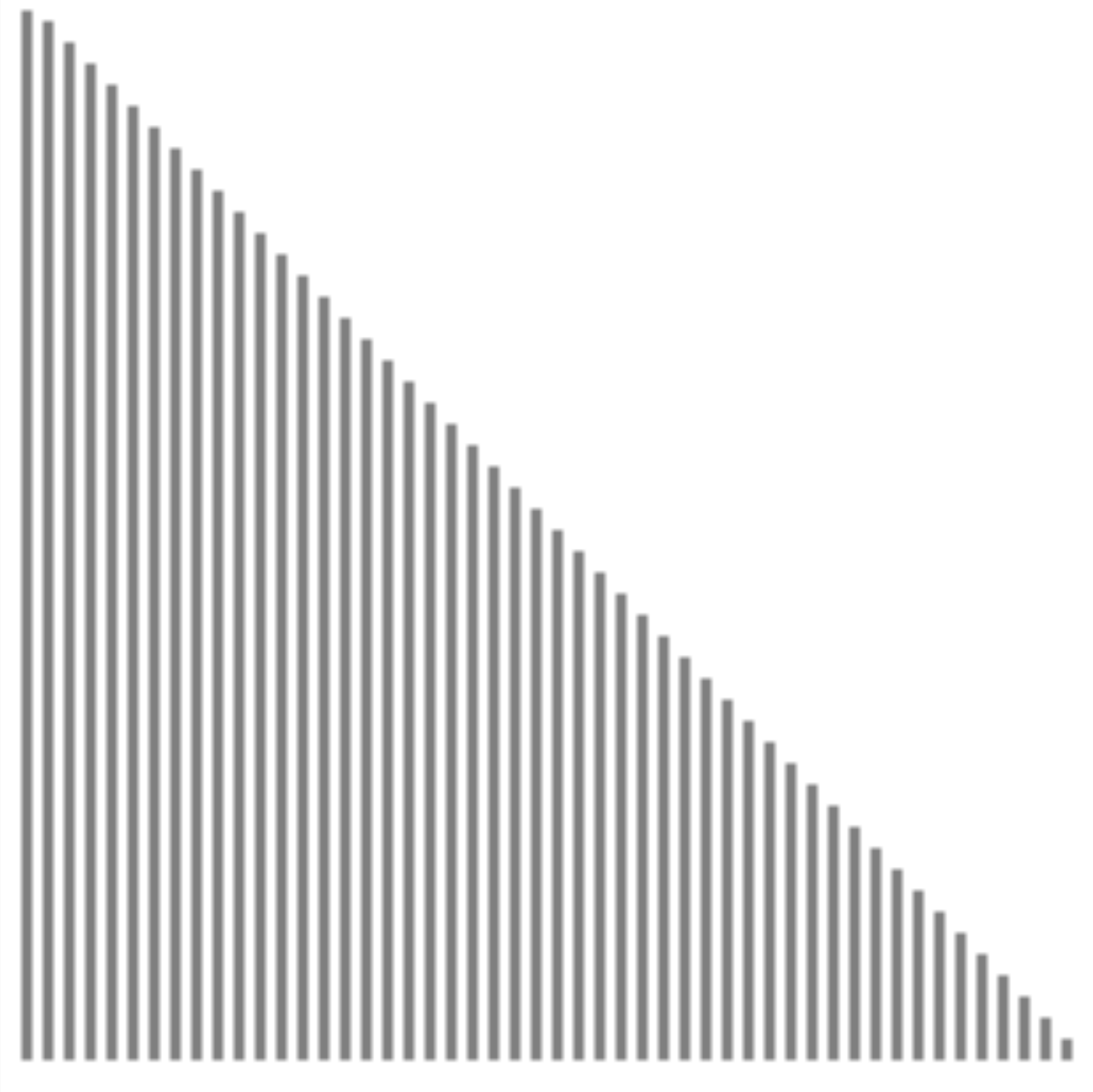
▬ in order

▬ current subarray

▬ not in order

# Mergesort: animation

50 reverse-sorted items



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

▲ algorithm position

▬ in order

▬ current subarray

▬ not in order

# 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 $\le n \lg n$ compares to sort any array of length $n$.

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

$$C(n) \le \underset{\substack{\uparrow \\ \text{left half}}}{C(\lceil n/2 \rceil)} + \underset{\substack{\uparrow \\ \text{right half}}}{C(\lfloor n/2 \rfloor)} + \underset{\substack{\uparrow \\ \text{merge}}}{n-1} \quad \text{for } n > 1\text{, with } C(1) = 0.$$

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

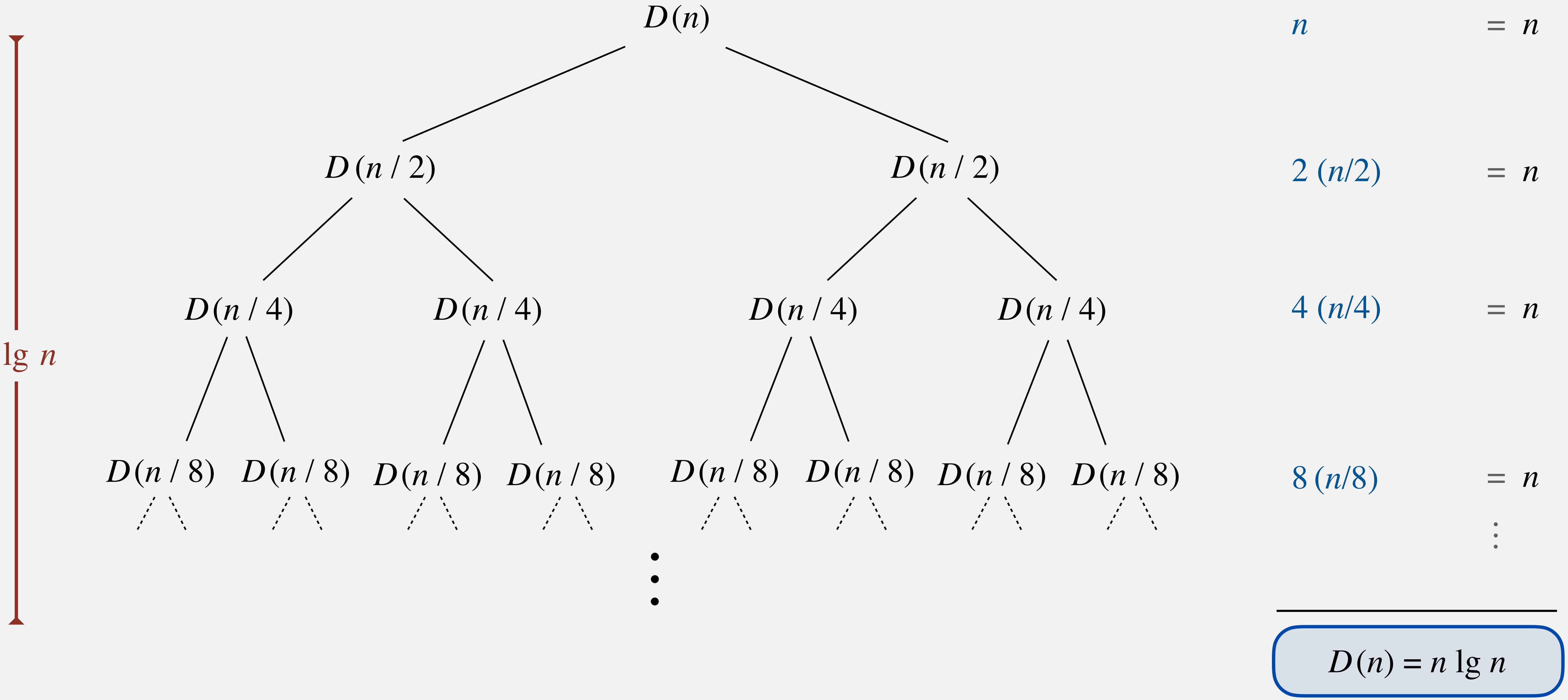
result holds for all $n$
(analysis cleaner in this case)

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

# 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]



$D(n) = n \lg n$

# Mergesort analysis

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

```
public static void f(int n)
{
    if (n == 0) return;
    f(n/2);
    f(n/2);
    linear(n);
}
```

solve two problems of half the size

do a linear amount of work

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

# Mergesort analysis: number of array accesses

Proposition. Mergesort uses $\le 6\, n \lg n$ array accesses to sort any array of length $n$.

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

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

[Rest of the proof is similar to the analysis of number of compares.]

# 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

"Essentially no extra memory"

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 $\sim \tfrac{1}{2} n$ instead of $n$.

Challenge 2 (very hard). In-place merge.

# 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);
   }

   public static void sort(Comparable[] a)
   {  /* as before */  }
}
```

Pf. 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++];
   }
}
```

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| $A_1$ | $A_2$ | $A_3$ | B | D | $A_4$ | $A_5$ | C | E | F | G |

Pf. Takes from left subarray if equal keys.

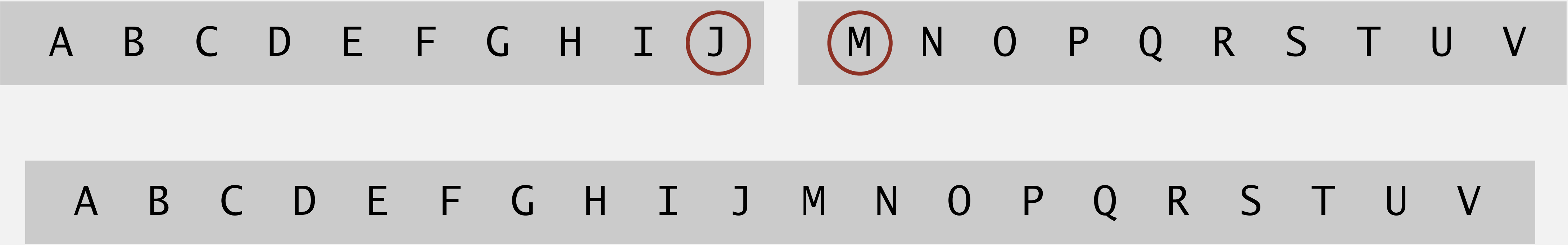
# Mergesort: practical improvements

Use insertion sort for small subarrays.

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

Stop if already sorted.

- Is largest item in first half ≤ 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

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

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.

Java 6: `Arrays.sort()` uses mergesort for sorting objects, with the above tricks.


Algorithms

Robert Sedgewick | Kevin Wayne

https://algs4.cs.princeton.edu

# 2.2 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, ....

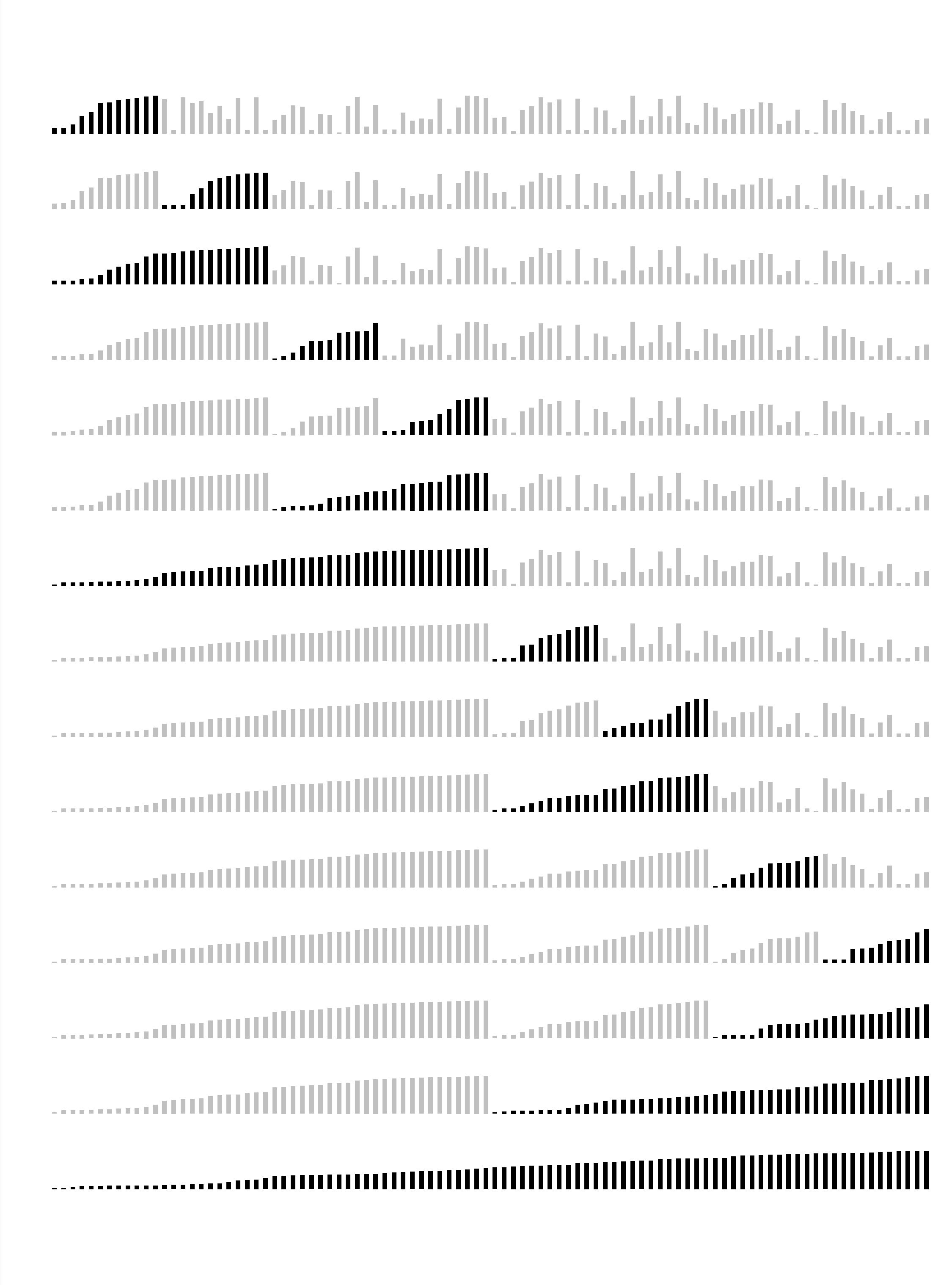
| | a[i] | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
| | M | E | R | G | E | S | O | R | T | E | X | A | M | P | L | E |
| sz = 1 | | | | | | | | | | | | | | | | |
| `merge(a,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

# Bottom-up mergesort: Java implementation

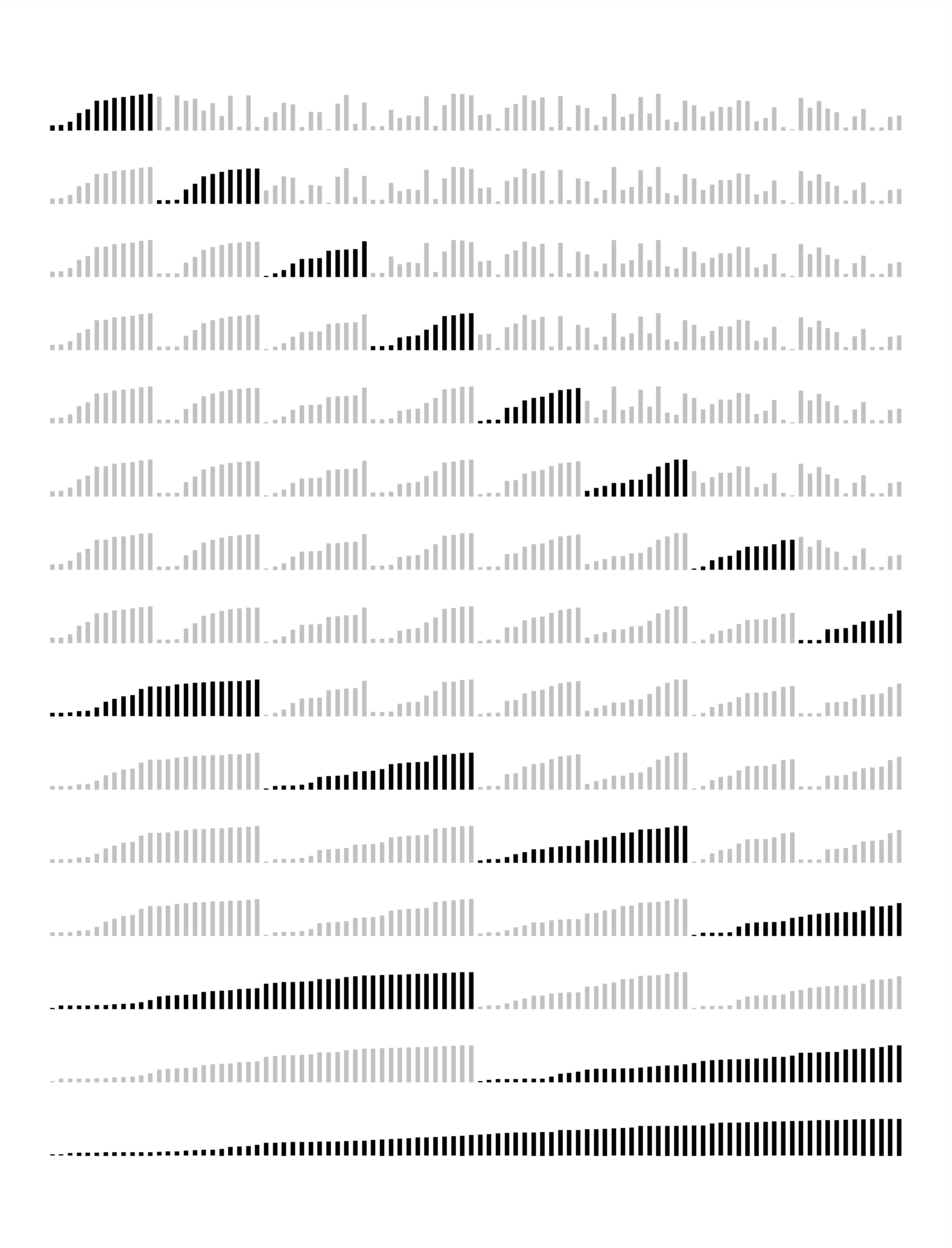
```java
public class MergeBU
{
   private static void merge(...)
   {  /* as before */  }

   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 for $n = 2^{20}$, top–down mergesort or bottom–up mergesort?**

**A.** Top-down (recursive) mergesort.

**B.** Bottom-up (non-recursive) mergesort.

**C.** No observable difference.

**D.** *I don't know.*

Hint: the answer depends on
concepts you’ll learn in COS 217.

# Sorting summary

| | inplace? | stable? | best | average | worst | remarks |
|---|---|---|---|---|---|---|
| **selection** | ✔ | | ½ $n^2$ | ½ $n^2$ | ½ $n^2$ | $n$ exchanges |
| **insertion** | ✔ | ✔ | $n$ | ¼ $n^2$ | ½ $n^2$ | use for small $n$ or partially ordered |
| **shell** | ✔ | | $n \log_3 n$ | ? | $c\, n^{3/2}$ | tight code; subquadratic |
| **merge** | | ✔ | ½ $n \lg n$ | $n \lg n$ | $n \lg n$ | $n \log n$ guarantee; stable |
| **?** | ✔ | ✔ | $n$ | $n \lg n$ | $n \lg n$ | holy sorting grail |

Algorithms

Robert Sedgewick | Kevin Wayne

https://algs4.cs.princeton.edu

# 2.2 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 $X$.

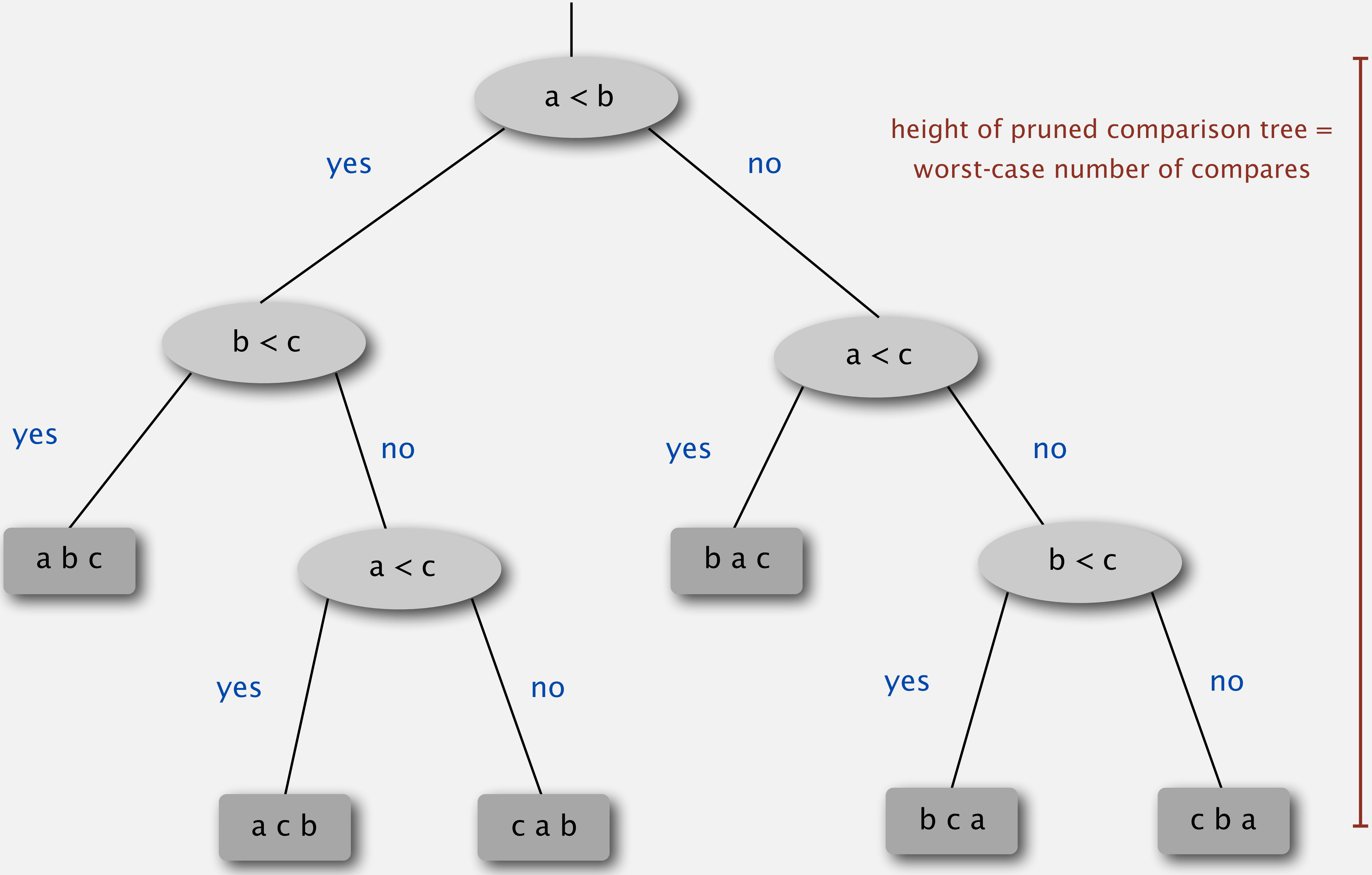
lower bound ~ upper bound

| | |
|---|---|
| **model of computation** | *comparison tree* |
| **cost model** | *# compares* |
| **upper bound** | $\sim n \lg n$ *from mergesort* |
| **lower bound** | ? |
| **optimal algorithm** | ? |

can access information
only through compares
(e.g., Java `Comparable` framework)

**complexity of sorting**

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



height of pruned comparison tree =
worst-case number of compares

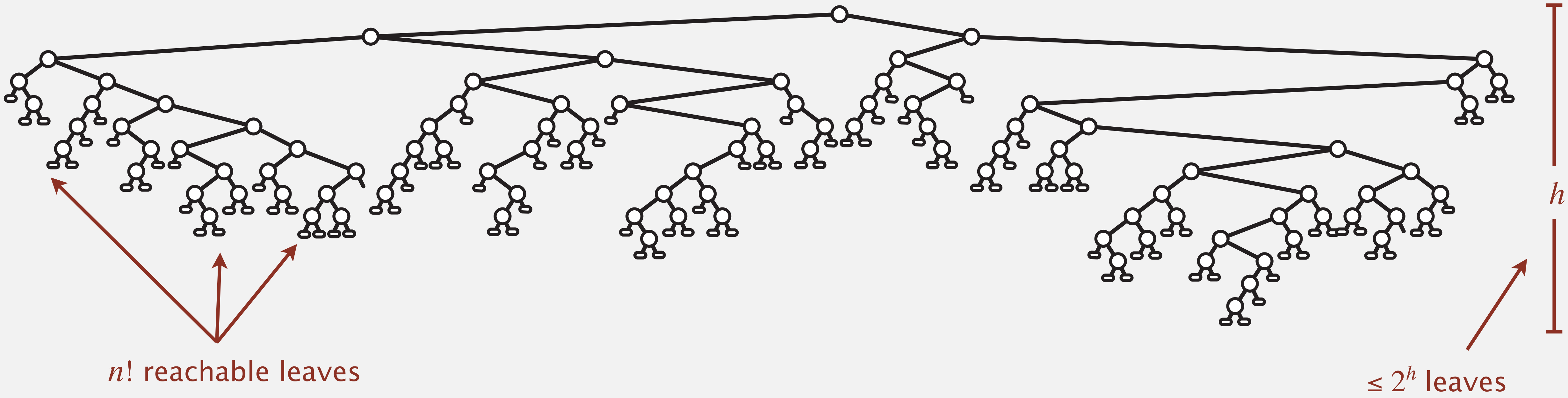
each reachable leaf corresponds to one (and only one) ordering;
exactly one reachable leaf for each possible ordering

# 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 number of compares = height $h$ of pruned comparison tree.
- Binary tree of height $h$ has $\leq 2^h$ leaves.
- $n!$ different orderings $\Rightarrow$ $n!$ reachable 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 number of compares = height $h$ of pruned comparison tree.
- Binary tree of height $h$ has $\leq 2^h$ leaves.
- $n!$ different orderings $\Rightarrow$ $n!$ reachable leaves.

$$\begin{aligned} 2^h &\geq \text{\# reachable leaves} = n! \\ \Rightarrow \quad h &\geq \lg(n!) \\ &\underset{\substack{\uparrow \\ \text{Stirling's formula}}}{\sim} n \lg n \end{aligned}$$

# 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** | *comparison tree* |
| **cost model** | *# compares* |
| **upper bound** | $\sim n \lg n$ |
| **lower bound** | $\sim n \lg n$ |
| **optimal algorithm** | *mergesort* |

**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.

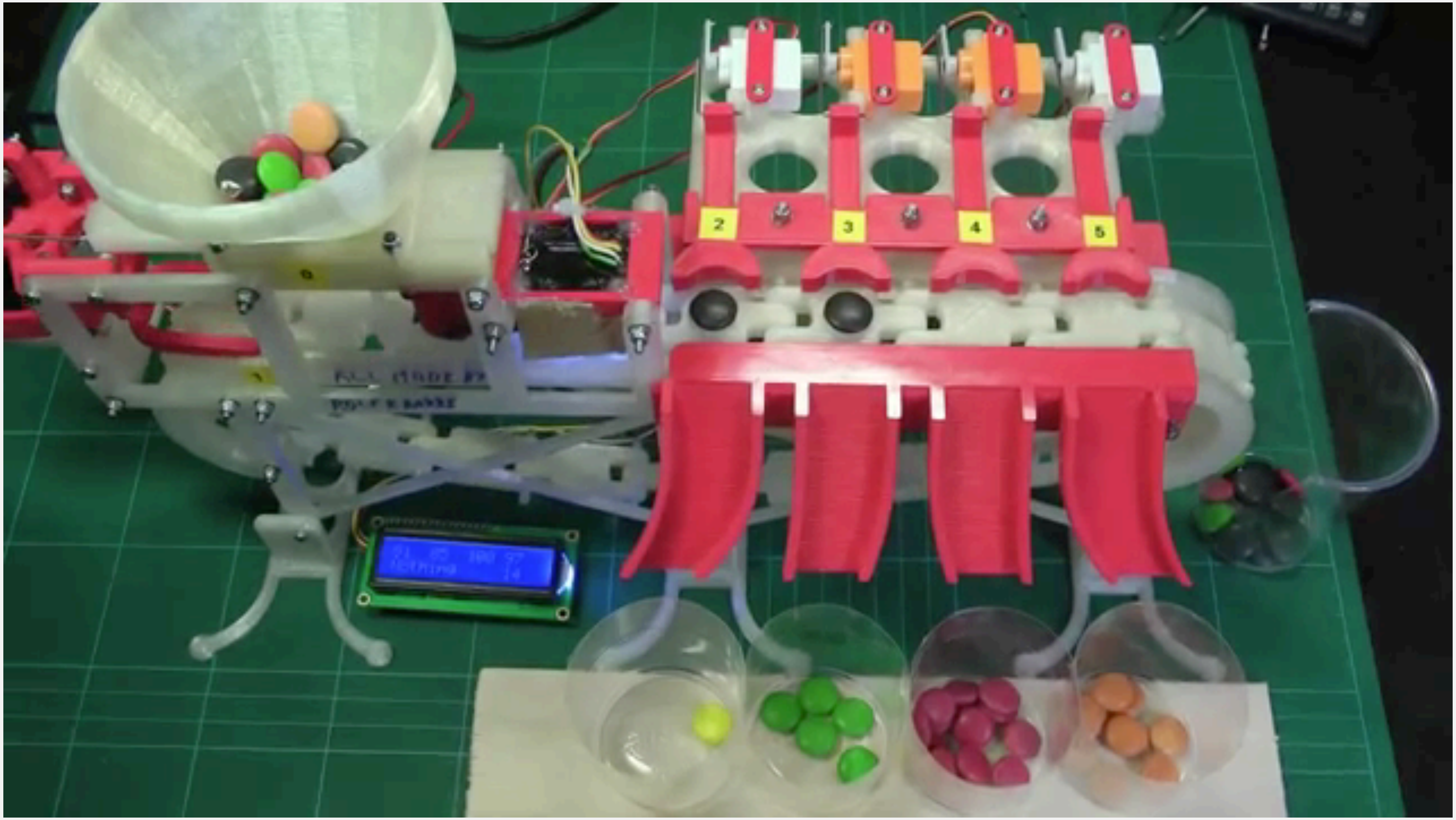
Lessons. Use theory as a guide.

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 sorter violate the sorting lower bound?



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

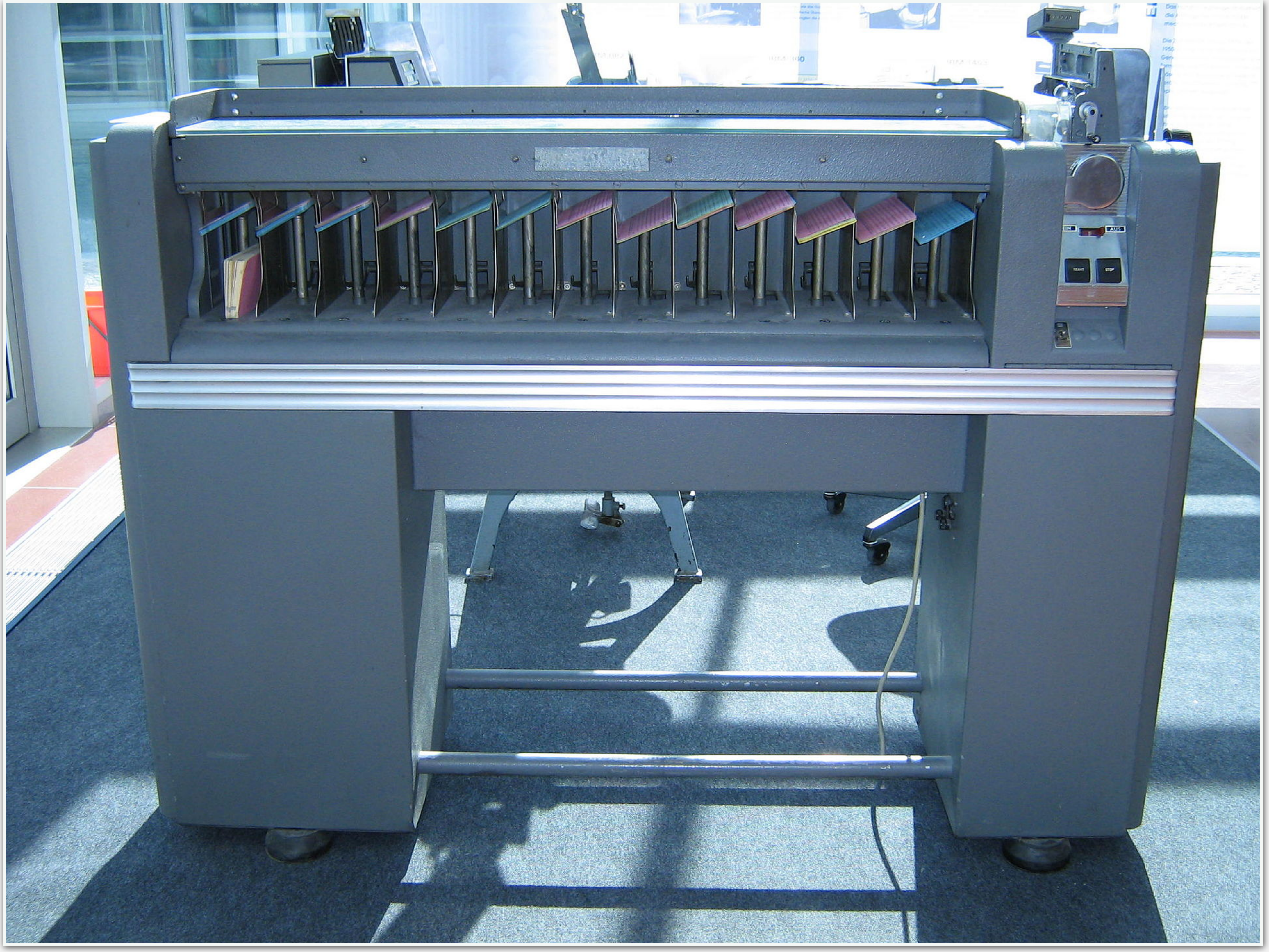
# Complexity results in context (continued)

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]

# A brief history of sorting: Hollerith census tabulator (1890s)

# IBM card sorter (1940s)



# Big O notation (and cousins)

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

# Understanding the notation

What's wrong with this statement? How would you correct it?

Any compare-based sorting algorithm must make at least $O(n \log n)$ compares in the worst case.

"At least $O(n \log n)$" is a nonsensical (but frequently heard) expression.

Correct: any compare-based sorting algorithm must make $\Omega(n \log n)$ compares in the worst case.

No need to say "at least $\Omega(n \log n)$" — that's implicit in the definition of $\Omega$.


Algorithms

Robert Sedgewick | Kevin Wayne

https://algs4.cs.princeton.edu

# 2.2 MERGESORT

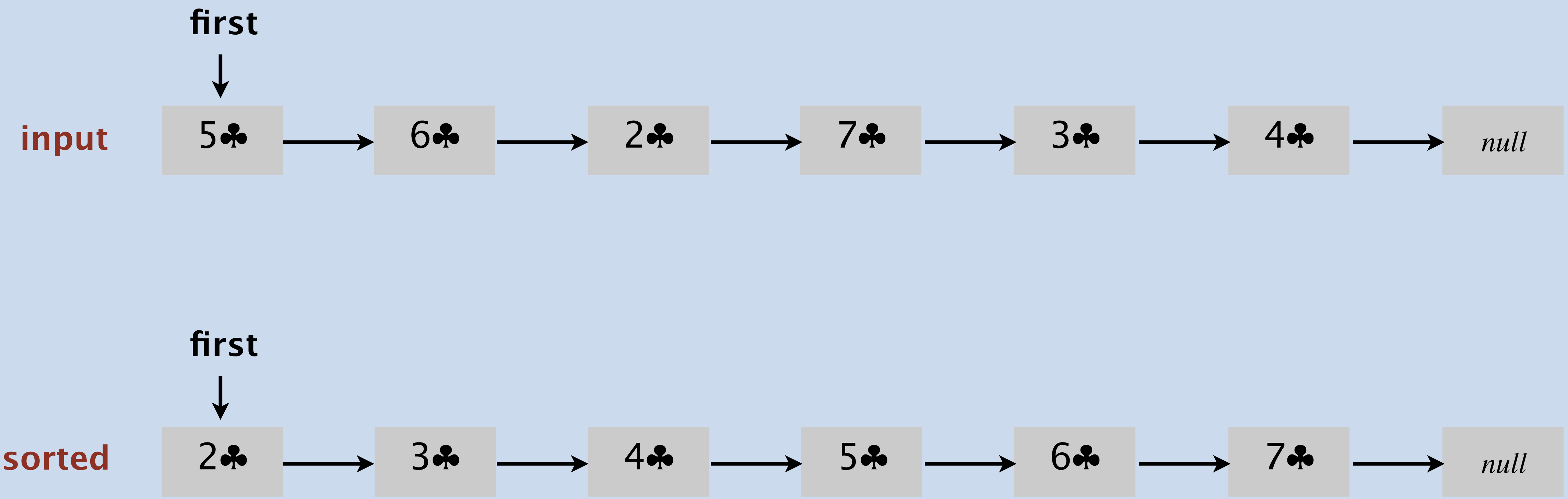
- *mergesort*
- *bottom-up mergesort*
- *sorting complexity*
- *divide-and-conquer*

# SORTING A LINKED LIST

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

Version 1. Linearithmic time, linear extra space.

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



first

**input** 5♣ → 6♣ → 2♣ → 7♣ → 3♣ → 4♣ → *null*

first

**sorted** 2♣ → 3♣ → 4♣ → 5♣ → 6♣ → 7♣ → *null*