



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

## 2.2 MERGESORT

---

- *mergesort*
- *bottom-up mergesort*
- *sorting complexity*
- *asymptotic notations*

# Two classic sorting algorithms: mergesort and quicksort

---

Critical components in our computational infrastructure.

Mergesort. [this lecture]



Quicksort. [next lecture]





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

## 2.2 MERGESORT

---

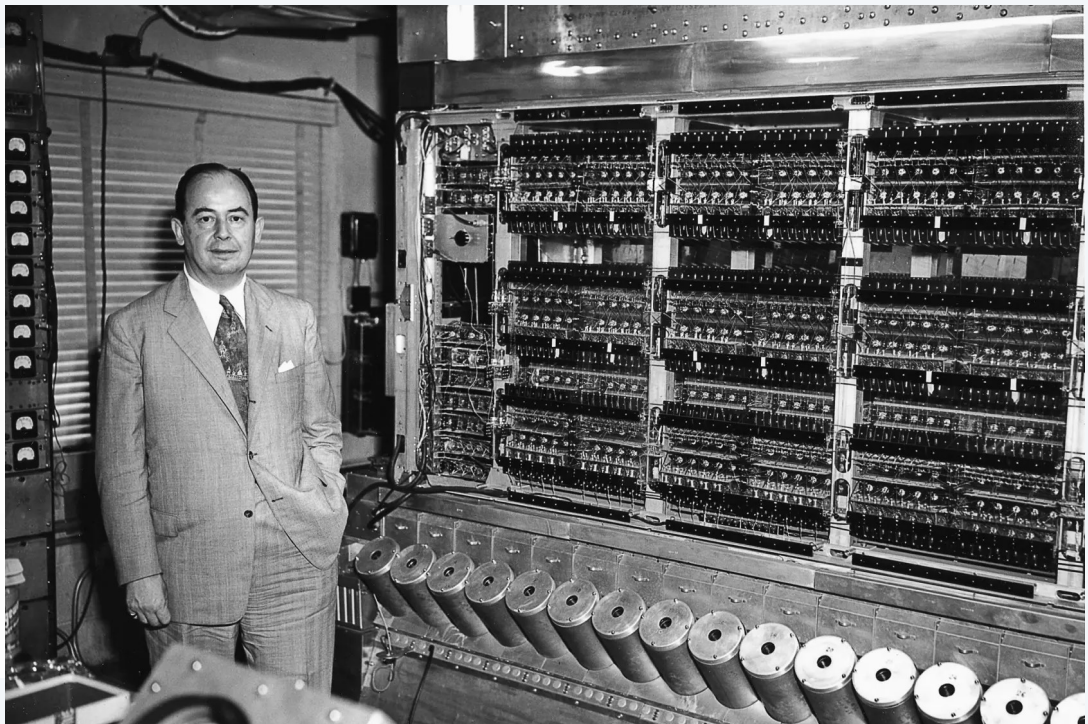
- *mergesort*
- *bottom-up mergesort*
- *sorting complexity*
- *asymptotic notations*

# Mergesort overview

## Basic plan.

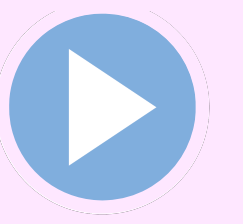
- Divide array into two halves.
- Recursively sort left half.
- Recursively sort right half.
- **Merge** two sorted halves.

First Draft  
of a  
Report on the  
EDVAC  
  
John von Neumann

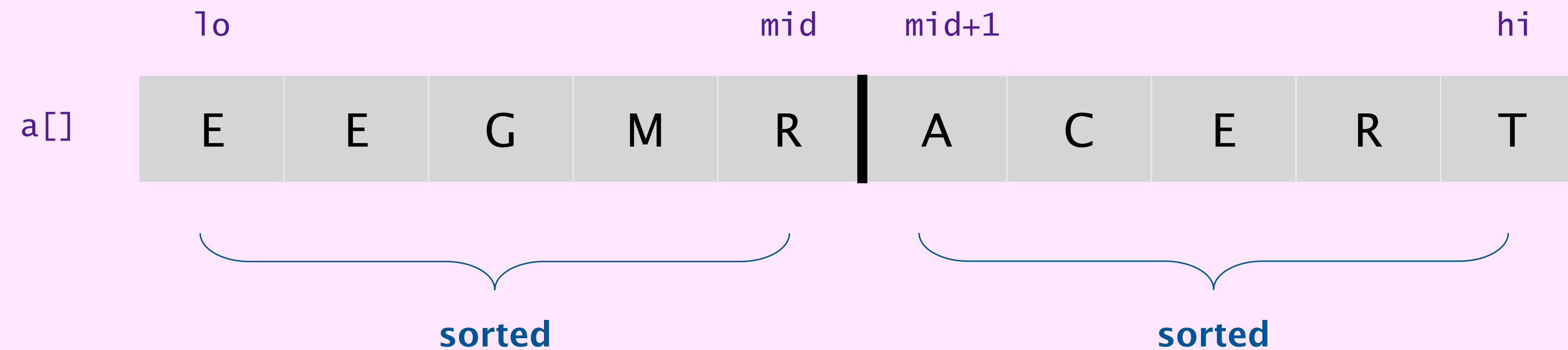


input	M	E	R	G	E	S	O	R	T	E	X	A	M	P	L	E
sort left half	E	E	G	M	O	R	R	S	T	E	X	A	M	P	L	E
sort right half	E	E	G	M	O	R	R	S	A	E	E	L	M	P	T	X
merge results	A	E	E	E	E	G	L	M	M	O	P	R	R	S	T	X

# 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

```
// precondition: a[lo..mid] and a[mid+1..hi] are sorted
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++];
    }
}
```

*merge subarrays a[lo..mid] and a[mid+1..hi]*

*copy*

*merge*

*left subarray exhausted*

*right subarray exhausted*

*select from right subarray*

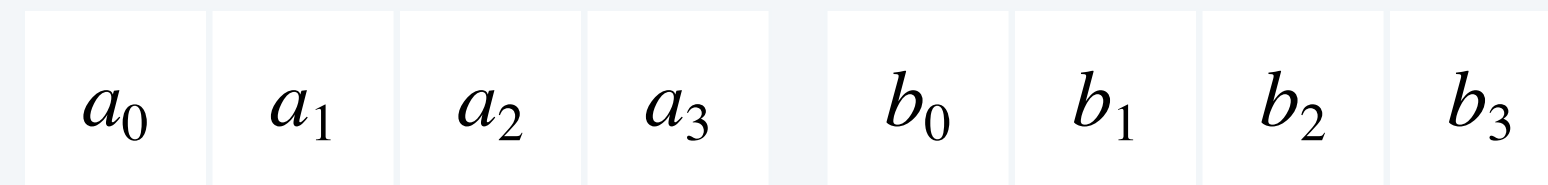
*select from left subarray*



# Mergesort overview

---

**Proposition.** The `merge()` method makes between  $n/2$  and  $n - 1$  calls to `less()` to merge two sorted subarrays each of length  $n/2$ .



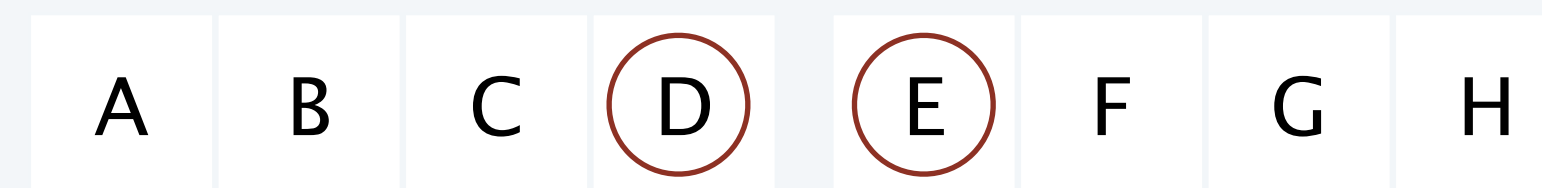
**Worst case.** Largest two elements are in different subarrays.

**Best case.** All elements in one subarray are larger than all elements in the other.

**worst-case input ( $n - 1$  compares)**



**best-case input ( $n/2$  compares)**



# 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);  
}
```

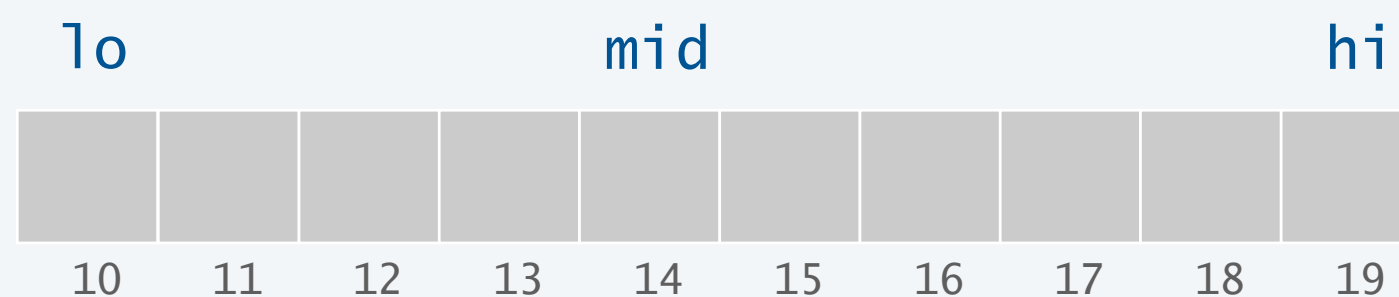
← *sort subarray a[lo..hi]*

← *sort left subarray*

← *sort right subarray*

← *merge left and right subarrays*

← *avoid allocating arrays  
within recursive function calls*



# Mergesort: trace

	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, <sup>lo</sup> 0, 0, <sup>hi</sup> 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



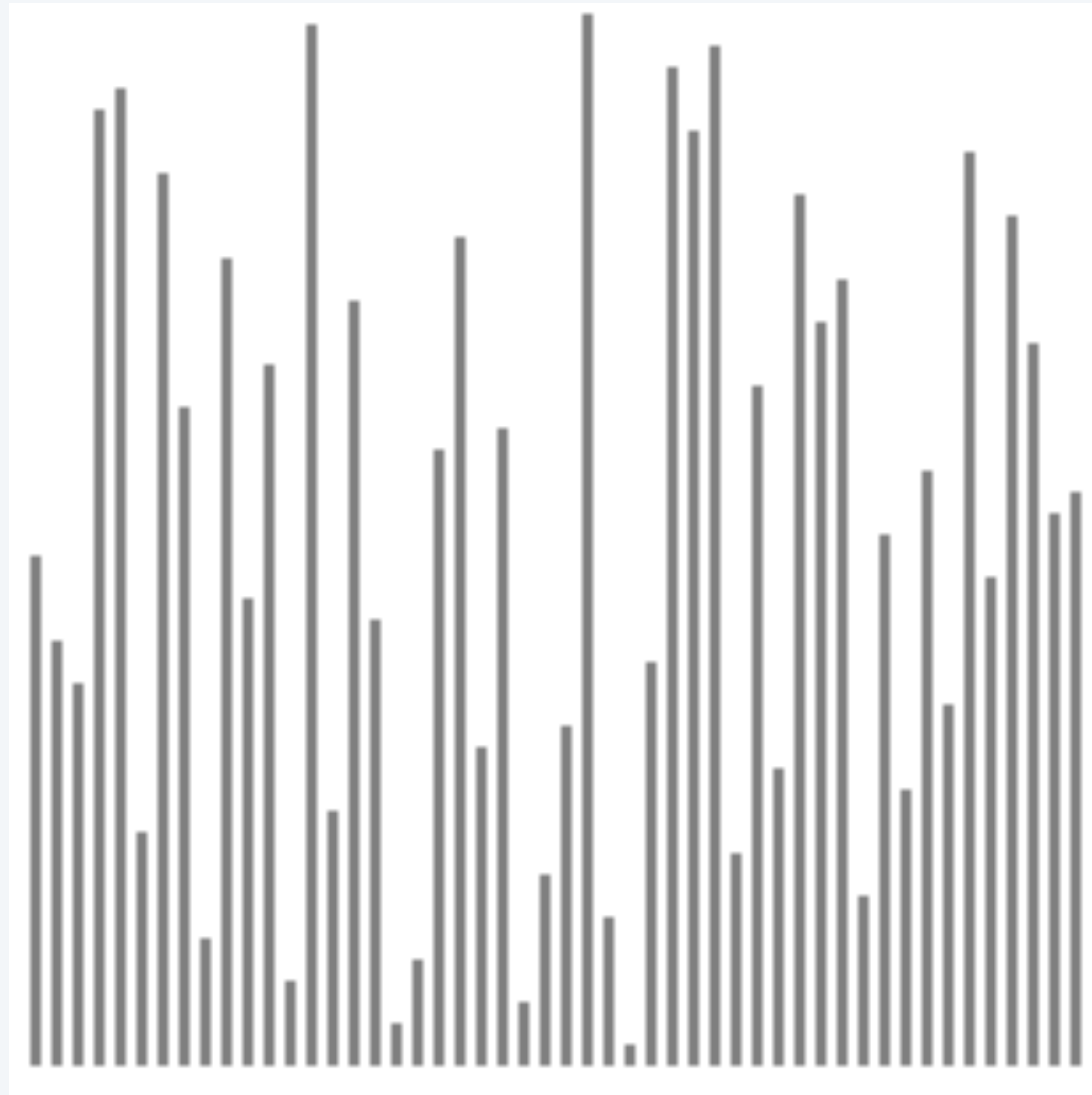
Which subarray lengths will arise when mergesorting an array of length  $n = 12$  ?

- 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 }
- E. Run-time exception.

# Mergesort: animation

---

50 random items



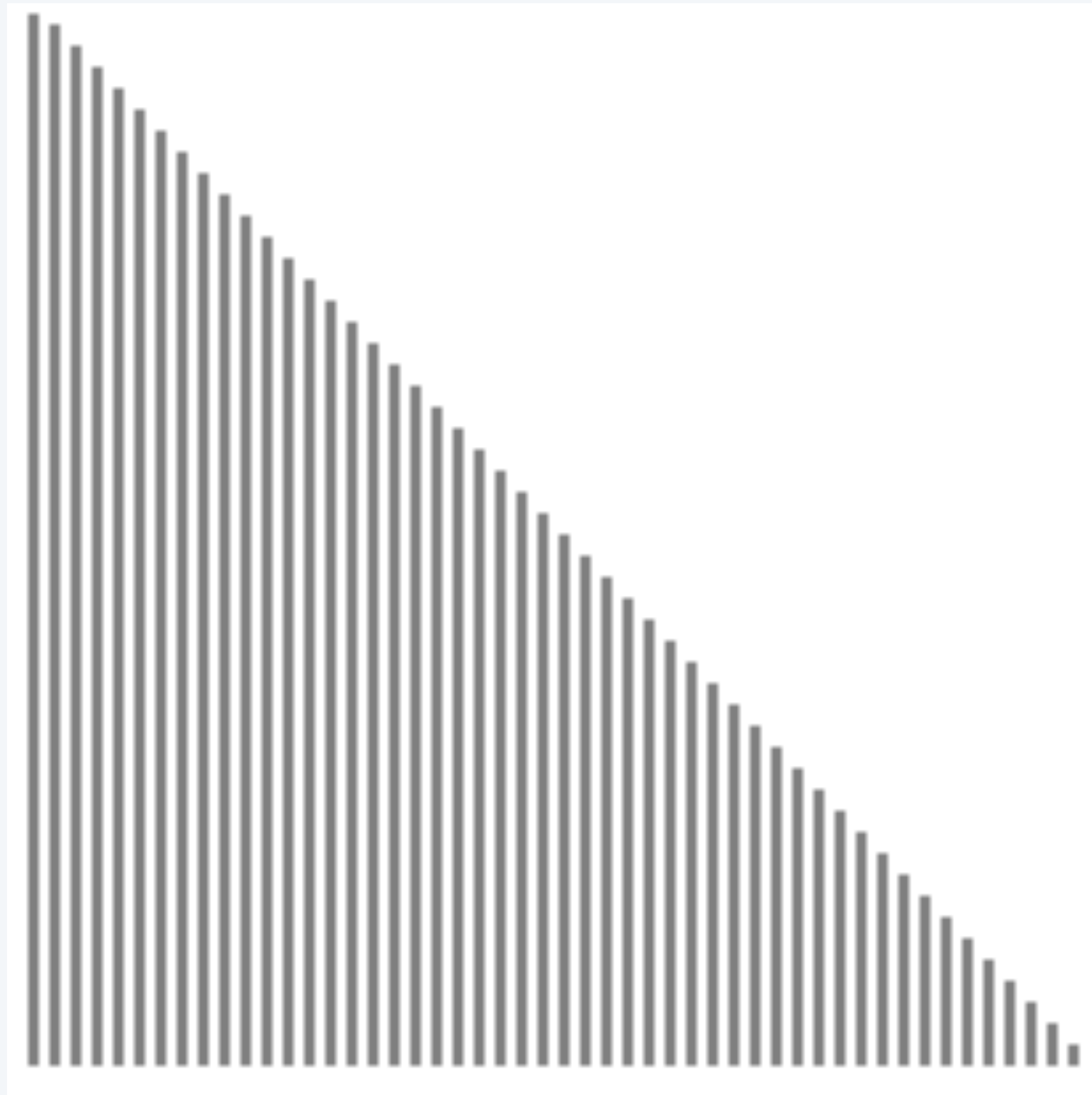
<https://www.toptal.com/developers/sorting-algorithms/merge-sort>

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


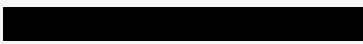
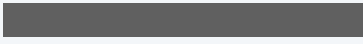
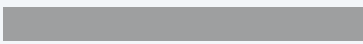
# Mergesort: animation

---

50 reverse-sorted items



<https://www.toptal.com/developers/sorting-algorithms/merge-sort>

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

# Insertion sort vs. mergesort: empirical analysis

## Running time estimates (approximate):

- Laptop executes  $10^8$  compares/second.
- Supercomputer executes  $10^{12}$  compares/second.



n	laptop	super
thousand	<i>instant</i>	<i>instant</i>
million	<i>2.8 hours</i>	<i>1 second</i>
billion	<i>317 years</i>	<i>1 week</i>

insertion sort

n	laptop	super
thousand	<i>instant</i>	<i>instant</i>
million	<i>1 second</i>	<i>instant</i>
billion	<i>18 minutes</i>	<i>instant</i>

mergesort

Bottom line. Great algorithms are better than supercomputers.

# Mergesort analysis: number of compares

---

**Proposition.** Mergesort uses  $\leq n \log_2 n$  compares to sort any array of length  $n$ .

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

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

↑  
*sort  
left half*

↑  
*sort  
right half*

↑  
*merge*

*proposition holds even when  $n$  is not a power of 2  
(but analysis cleaner in this case)*



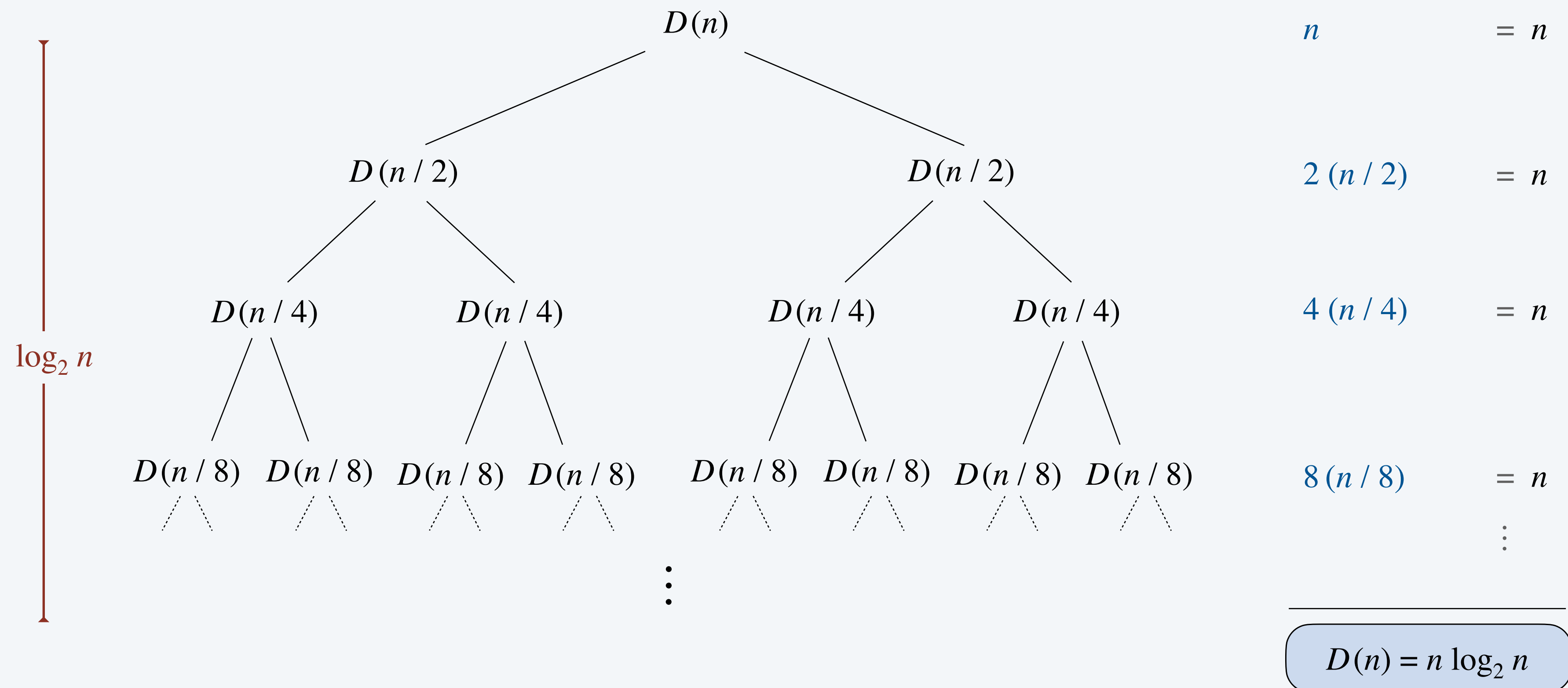
**For simplicity.** Assume  $n$  is a power of 2 and solve this recurrence:

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

# Divide-and-conquer recurrence

**Proposition.** If  $D(n)$  satisfies  $D(n) = 2D(n/2) + n$  for  $n > 1$ , with  $D(1) = 0$ , then  $D(n) = n \log_2 n$ .

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



# Mergesort analysis: number of array accesses

---

**Proposition.** Mergesort makes  $\Theta(n \log n)$  array accesses.

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

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

**Divide-and-conquer.** 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  $\Theta(n)$  work*

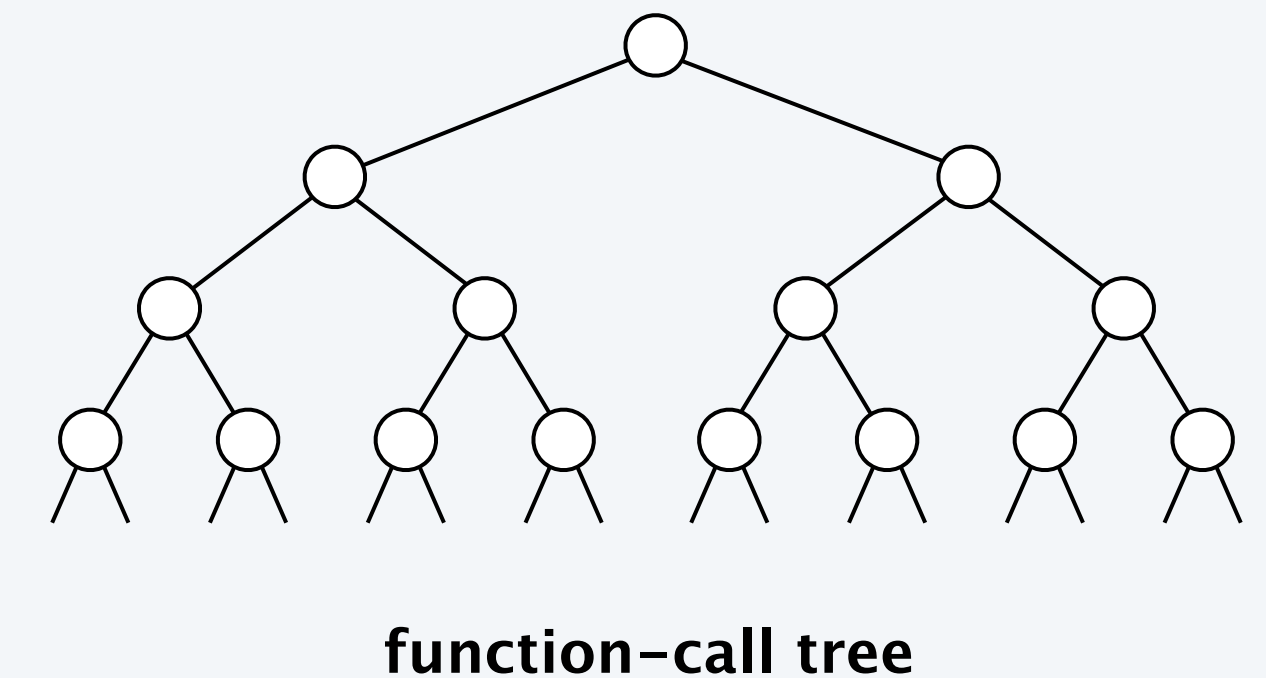
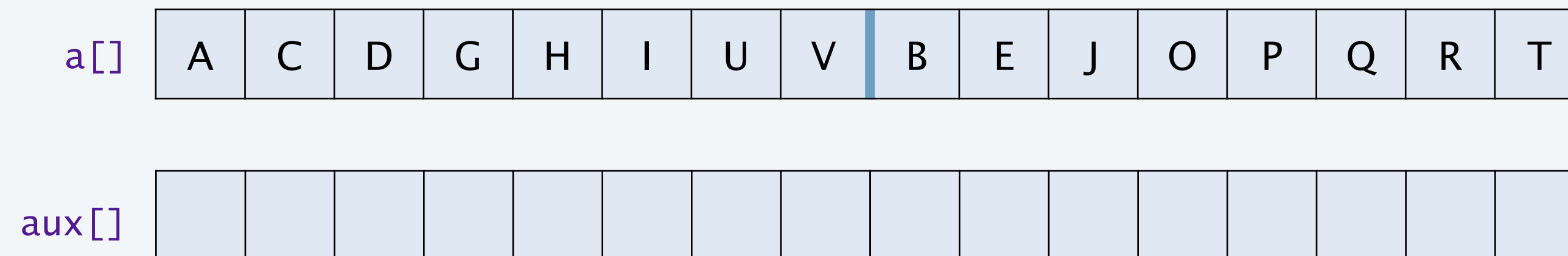
**Famous examples.** FFT, closest pair, hidden-line removal, Kendall-tau distance, ...

# Mergesort analysis: memory

**Proposition.** Mergesort uses  $\Theta(n)$  extra space.

**Pf.**

- The length of the `aux[]` array is  $n$ .
- The max depth of the function-call stack (for recursion) is  $\log_2 n$ .



**Def.** A sorting algorithm is **in-place** if it uses  $\Theta(\log n)$  extra space (or less). ← essentially negligible  
(includes memory for any recursive calls)

**Ex.** Insertion sort and selection sort.

**Challenge 1 (not hard).** Merge using an `aux[]` array of length  $\frac{1}{2}n$  (instead of  $n$ ).

**Challenge 2 (very hard).** Merge using only  $\Theta(\log n)$  or  $\Theta(1)$  extra space. [Kronrod 1969]



Consider the following **modified** version of mergesort.

How much total memory is allocated (and deallocated) over all recursive calls?

- A.  $\Theta(n)$
- B.  $\Theta(n \log n)$
- C.  $\Theta(n^2)$
- D.  $\Theta(2^n)$

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

*allocates array in  
recursive method*

# Mergesort: practical improvement

## Use insertion sort for small subarrays.

- Mergesort has too much overhead for tiny subarrays.
- Cutoff to insertion sort for  $\approx 12$  items. ← *Java system sort uses cutoff value = 7*

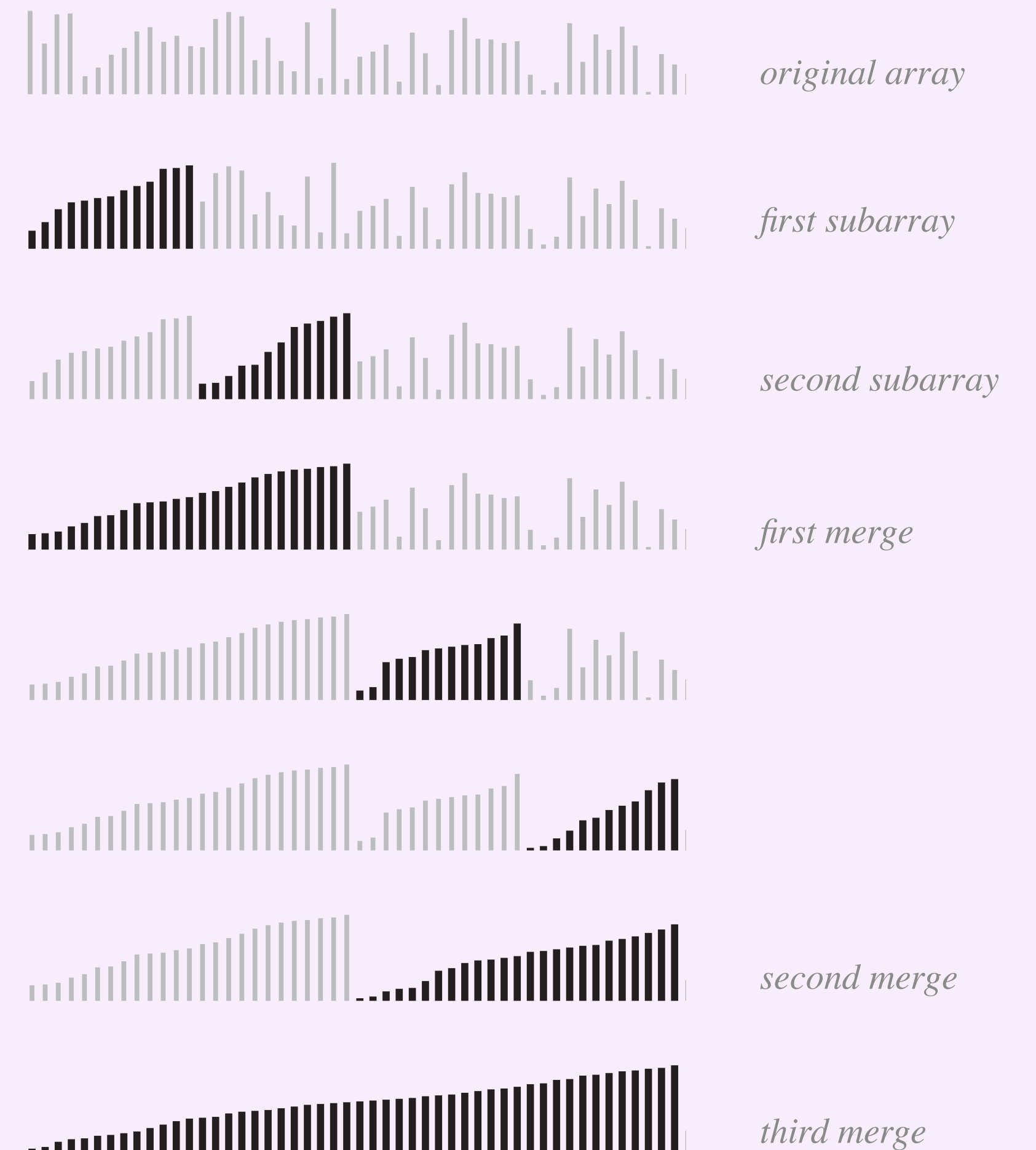
```
private static void sort(...) {
```

```
    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);
```

```
}
```

*makes mergesort  
about 20% faster*



**mergesort with cutoff to insertion sort**  
( $n = 48$ , cutoff = 12)



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

## 2.2 MERGESORT

---

- *mergesort*
- *bottom-up mergesort*
- *sorting complexity*
- *asymptotic notations*


# Bottom-up mergesort

## Basic plan.

- Pass through array, merging subarrays of length 1.
- Repeat for subarrays of length 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, 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	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	A	E	E	E	E	G	L	M	M	O	P	R	R	S	T	X

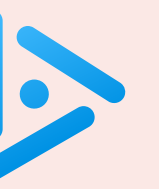
# Bottom-up mergesort: Java implementation

```
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)  length of subarrays  
to merge  
            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));  
    }  
}
```

mid                      hi

**Proposition.** At most  $n \log_2 n$  compares;  $\Theta(n)$  extra space.

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



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 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 (2002)

---

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

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.

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.

...



**Tim Peters**

**Consequence.** Only  $\Theta(n)$  compares to sort many arrays with pre-existing order.

**Widely used.** Python, Java, Android, Swift, Rust, V8 JavaScript, ...



# Envisage: Engineering Virtualized Services

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.

# Timsort bug (May 2018)



JDK / [JDK-8203864](#)

## Execution error in Java's Timsort

### Details

Type:	Bug
Status:	<b>RESOLVED</b>
Priority:	P3
Resolution:	Fixed
Affects Version/s:	None
Fix Version/s:	11
Component/s:	core-libs
Labels:	None
Subcomponent:	java.util:collections
Introduced In Version:	6
Resolved In Build:	b20

### Description

Carine Pivoteau wrote:  
While working on a proper complexity analysis of the algorithm, we realised that there was an error in the last paper reporting such a bug (<http://envisage-project.eu/wp-content/uploads/2015/02/sorting.pdf>). This implies that the correction implemented in the Java source code (changing Timsort stack size) is wrong and that it is still possible to make it break. This is explained in full details in our analysis: <https://arxiv.org/pdf/1805.08612.pdf>.  
We understand that coming upon data that actually causes this error is very unlikely, but we thought you'd still like to know and do something about it. As the authors of the previous article advocated for, we strongly believe that you should consider modifying the algorithm as explained in their article (and as was done in Python) rather than trying to fix the stack size.

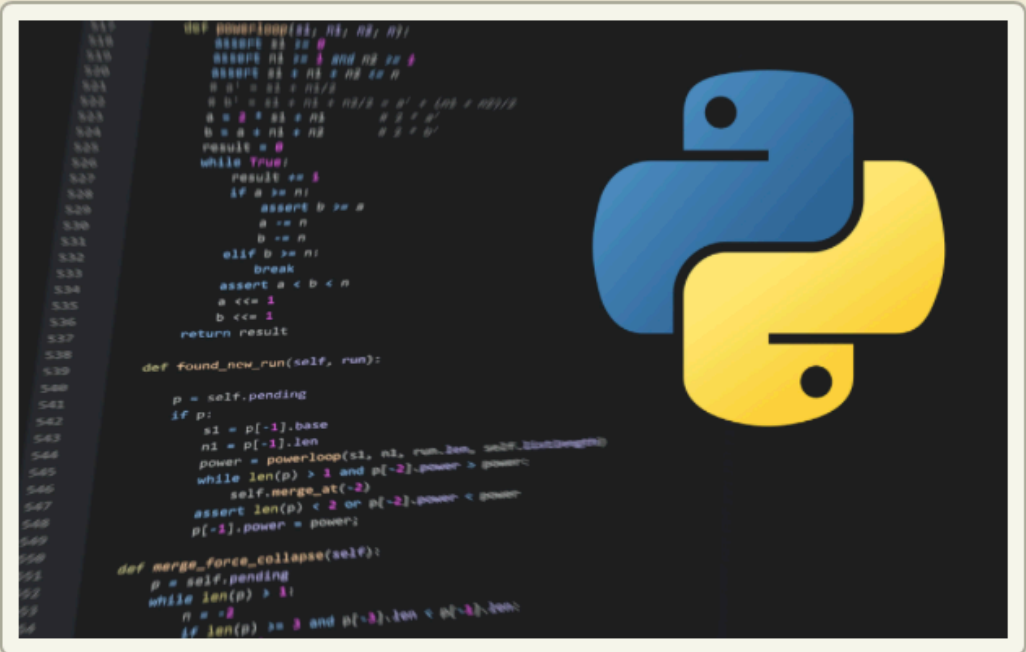
Algorithmic progress is ongoing. A version of Timsort that optimizes order of merges.

# Powersort in official Python 3.11 release

web teaching algorithms 24 Oct 2022

Our sorting method *Powersort* is used as default `list.sort()` algorithm in CPython, the reference implementation of the Python programming language.

See my *PyCon US* talk for the full story. Here's the entry from the official *Python changelog*:



*bpo-34561: List sorting now uses the merge-ordering strategy from Munro and Wild's `powersort()`. Unlike the former strategy, this is provably near-optimal in the entropy of the distribution of run lengths. Most uses of `list.sort()` probably won't see a significant time difference, but may see significant improvements in cases where the former strategy was exceptionally poor. However, as these are all fast linear-time approximations to a problem that's inherently at best quadratic-time to solve truly optimally, it's also possible to contrive cases where the former strategy did better.*



# Sorting summary

	in-place?	stable?	best	typical	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 sorted
merge		✓	$\frac{1}{2} n \log_2 n$	$n \log_2 n$	$n \log_2 n$	$\Theta(n \log n)$ guarantee; stable
timsort		✓	$n$	$n \log_2 n$	$n \log_2 n$	improves mergesort when pre-existing order
?	✓	✓	$n$	$n \log_2 n$	$n \log_2 n$	holy sorting grail

number of compares to sort an array of  $n$  elements (tilde notation)



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



## 2.2 MERGESORT


---

- *mergesort*
- *bottom-up mergesort*
- *sorting complexity*
- *asymptotic notations*

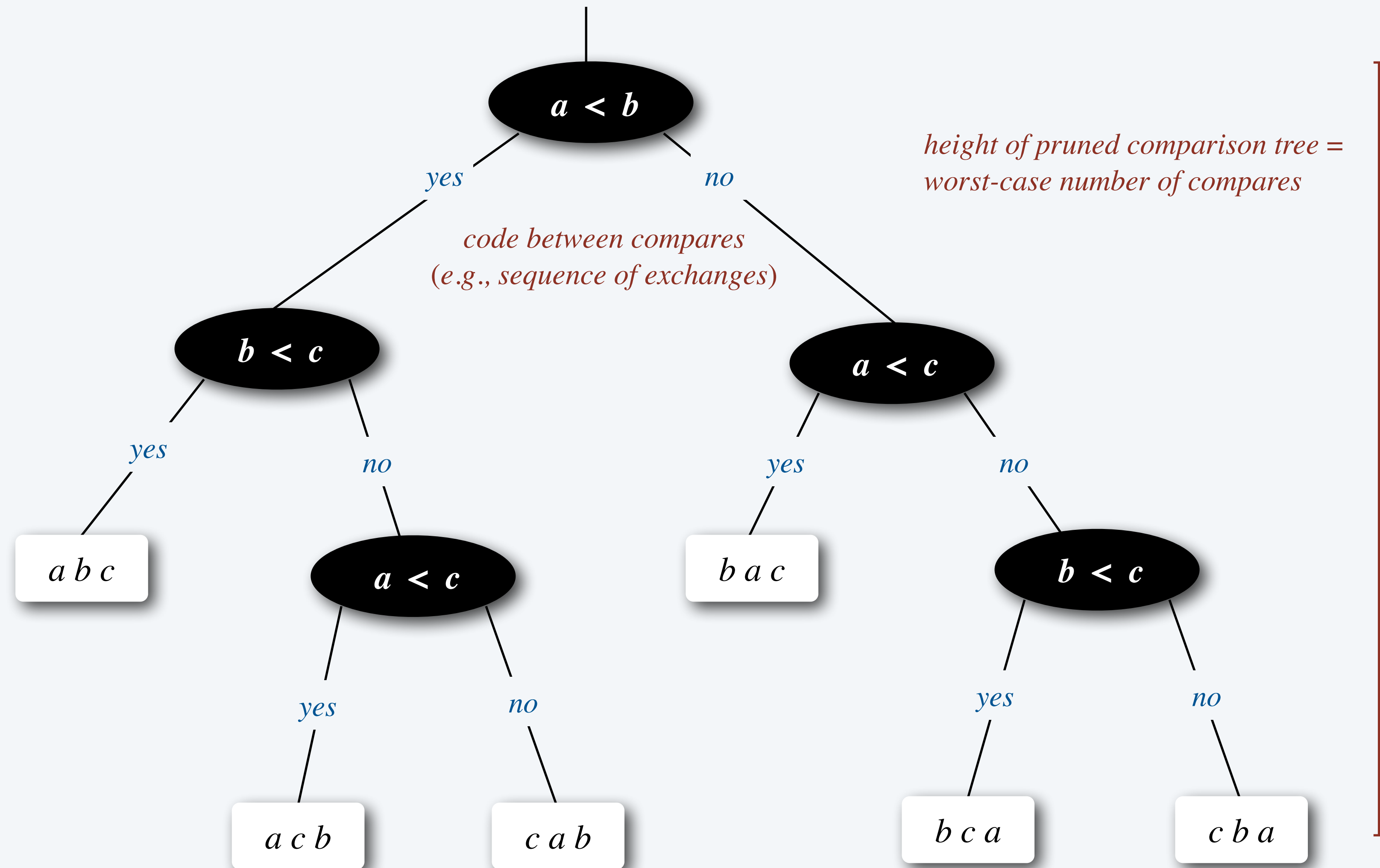
# Computational complexity

A framework to study efficiency of algorithms for solving a particular problem  $X$ .

term	description	example ( $X = \text{sorting}$ )
model of computation	<i>specifies memory and primitive operations</i>	comparison tree  <i>can gain knowledge about input only through pairwise compares (e.g., Java's Comparable framework)</i>
cost model	<i>primitive operation counts</i>	# compares
upper bound	<i>cost guarantee provided by <b>some</b> algorithm for a problem</i>	$\sim n \log_2 n$  <i>from mergesort</i>
lower bound	<i>proven limit on cost guarantee for <b>all</b> algorithms for a problem</i>	?
optimal algorithm	<i>algorithm with <b>best</b> possible cost guarantee for a problem</i>	?

 *lower bound ~ upper bound*

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



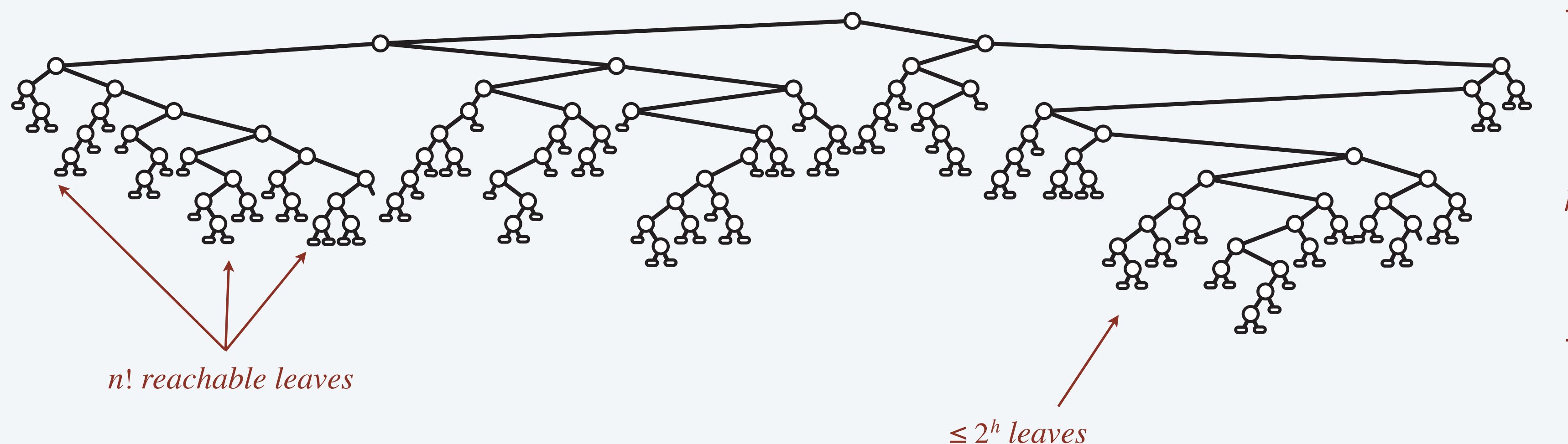
# Compare-based lower bound for sorting

**Proposition.** In the worst case, any compare-based sorting algorithm must make at least  $\log_2(n!) \sim n \log_2 n$  compares.

**Pf.**

*lower bound holds even  
for this special kind of input*

- Assume array consists of  $n$  distinct values.
- $n!$  different orderings  $\implies n!$  reachable leaves.
- Worst-case number of compares = height  $h$  of pruned comparison tree.
- Binary tree of height  $h$  has  $\leq 2^h$  leaves.



# Compare-based lower bound for sorting

---

**Proposition.** In the worst case, any compare-based sorting algorithm must make at least  $\log_2(n!) \sim n \log_2 n$  compares.

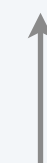
**Pf.**

- Assume array consists of  $n$  distinct values.
- $n!$  different orderings  $\implies n!$  reachable leaves.
- Worst-case number of compares = height  $h$  of pruned comparison tree.
- Binary tree of height  $h$  has  $\leq 2^h$  leaves.

$$2^h \geq \# \text{ reachable leaves} = n!$$

$$\implies h \geq \log_2(n!)$$

$$\sim n \log_2 n$$



*logarithmic sum  
(Stirling's formula)*

# Computational complexity

A framework to study efficiency of algorithms for solving a particular problem  $X$ .

term	description	example ( $X = \text{sorting}$ )
model of computation	<i>specifies memory and primitive operations</i>	comparison tree
cost model	<i>primitive operation counts</i>	# compares
upper bound	<i>cost guarantee provided by <b>some</b> algorithm for a problem</i>	$\sim n \log_2 n$
lower bound	<i>proven limit on cost guarantee for <b>all</b> algorithms for a problem</i>	$\sim n \log_2 n$
optimal algorithm	<i>algorithm with <b>best</b> possible cost guarantee for a problem</i>	mergesort

First goal of algorithm design: optimal algorithms.

# Computational complexity results in context

---

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

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



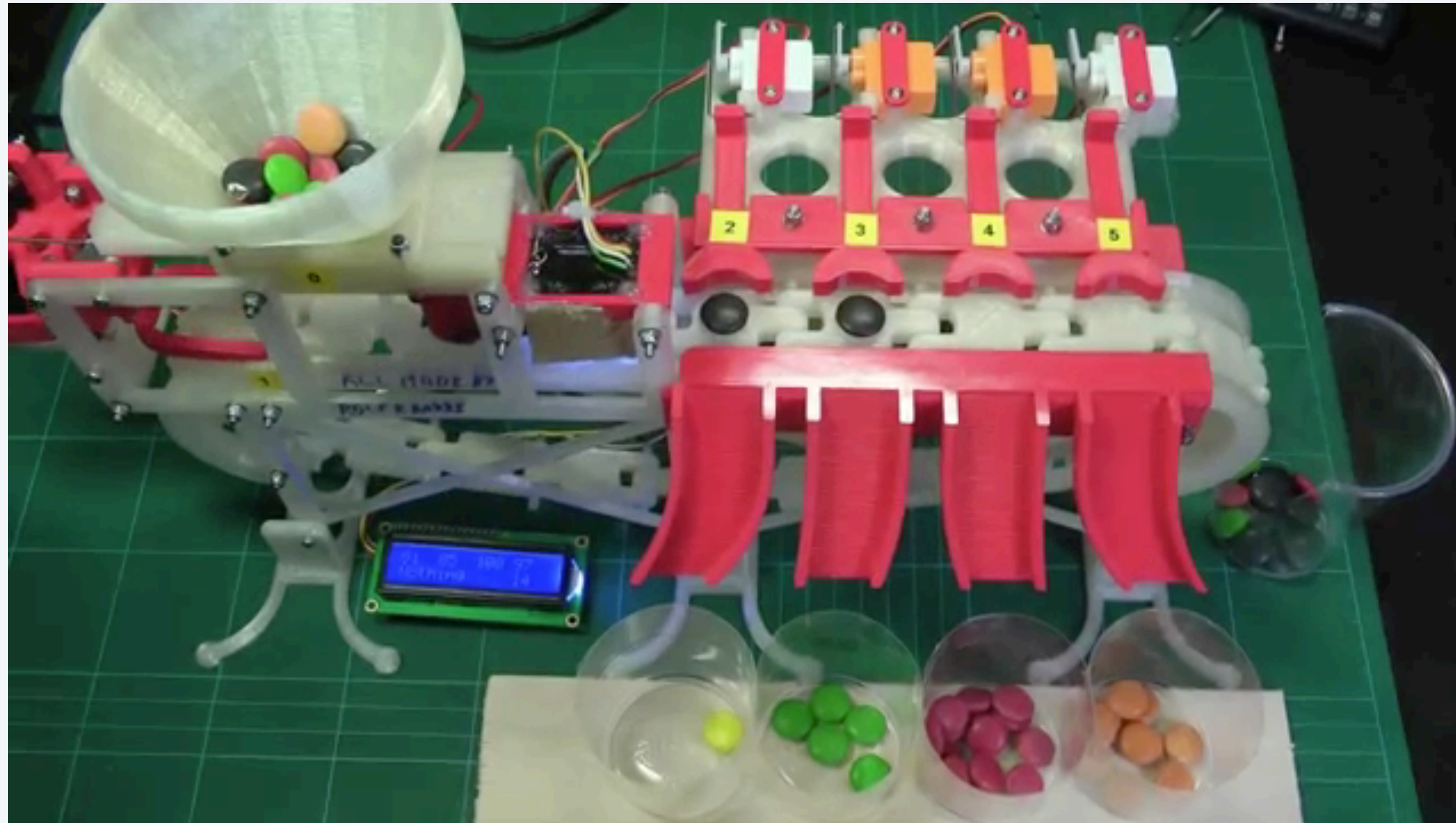
**Lesson.** Use theory as a guide.

**Ex 1.** Design sorting algorithm that makes  ~~$\sim \frac{1}{2} n \log_2 n$  compares in worst case?~~

**Ex 2.** Design sorting algorithm that makes  $\Theta(n \log n)$  compares and uses  $\Theta(1)$  extra space.



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



## Complexity results in context (continued)

---

Lower bound may not hold if the algorithm can exploit:

- The initial order of the input array.

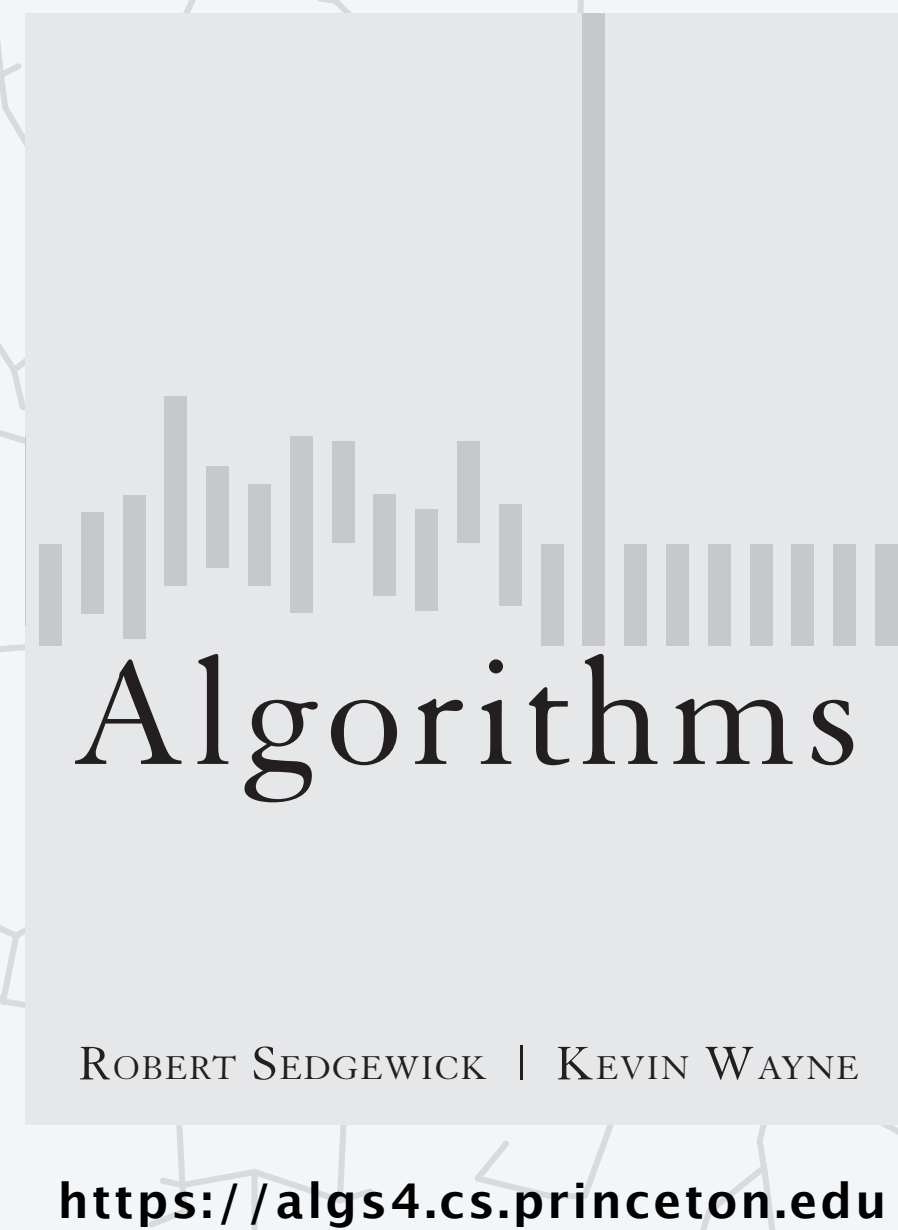
Ex: insertion sort makes only  $\Theta(n)$  compares on partially sorted arrays.

- The distribution of key values.

Ex: 3-way quicksort makes only  $\Theta(n)$  compares on arrays with only a few distinct key values. [next lecture]

- The representation of the keys.

Ex: MSD radix sort takes linear time to sort integers (or strings); it accesses the keys via individual digits (or characters), not key compares.



## 2.2 MERGESORT

---

- *mergesort*
- *bottom-up mergesort*
- *sorting complexity*
- *asymptotic notations*

# Asymptotic notations

Warning: many programmers misuse  $O$  to mean  $\Theta$ .

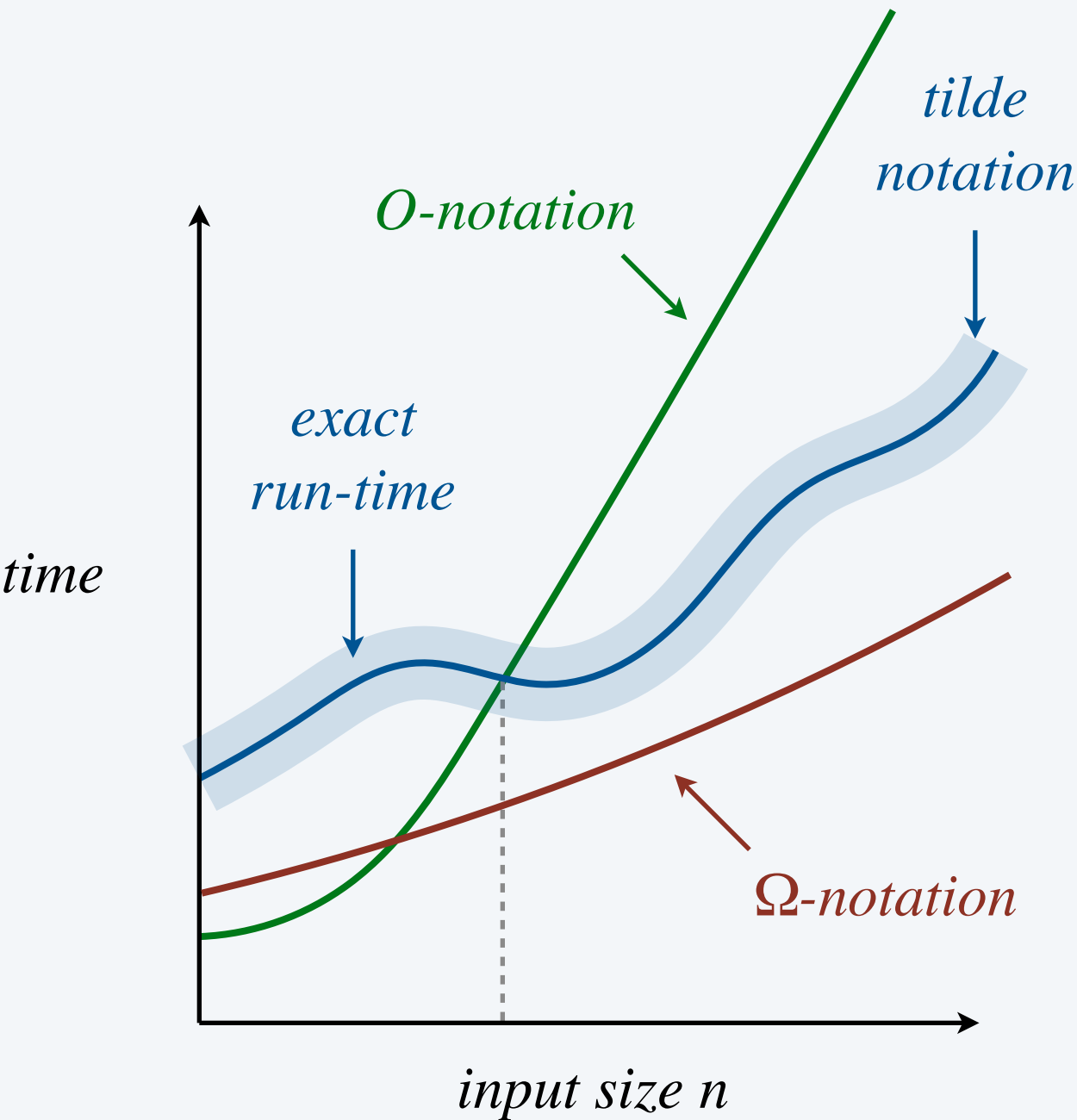
notation	provides	example	shorthand for
<b>tilde</b> ( $\sim$ )	<i>leading term</i>	$\sim \frac{1}{2} n^2$	$\frac{1}{2} n^2$ $\frac{1}{2} n^2 + 3n + 22$ $\frac{1}{2} n^2 + n \log_2 n$
<b>big Theta</b> ( $\Theta$ )	<i>order of growth</i>	$\Theta(n^2)$	$\frac{1}{2} n^2$ $7 n^2 + n^{\frac{1}{2}}$ $5 n^2 - 3n$
<b>big O</b> ( $O$ )	<i>upper bound</i>	$O(n^2)$	$10 n^2$ $22 n$ $\log_2 n$
<b>big Omega</b> ( $\Omega$ )	<i>lower bound</i>	$\Omega(n^2)$	$\frac{1}{2} n^2$ $n^3 + 3n$ $2^n$

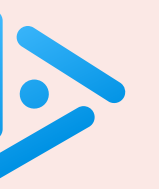
*ignore lower-order terms*

*also ignore leading coefficient*

$\Theta(n^2)$  or smaller

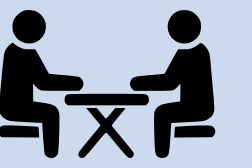
$\Theta(n^2)$  or larger





Which of the following correctly describes the function  $f(n) = 3n^2 + 30n$  ?

- A.  $\sim n^2$
- B.  $\Theta(n)$
- C.  $O(n^3)$
- D. *All of the above.*
- E. *None of the above.*



**Interviewer.** Give a formal description of the sorting lower bound for sorting arrays of  $n$  elements.



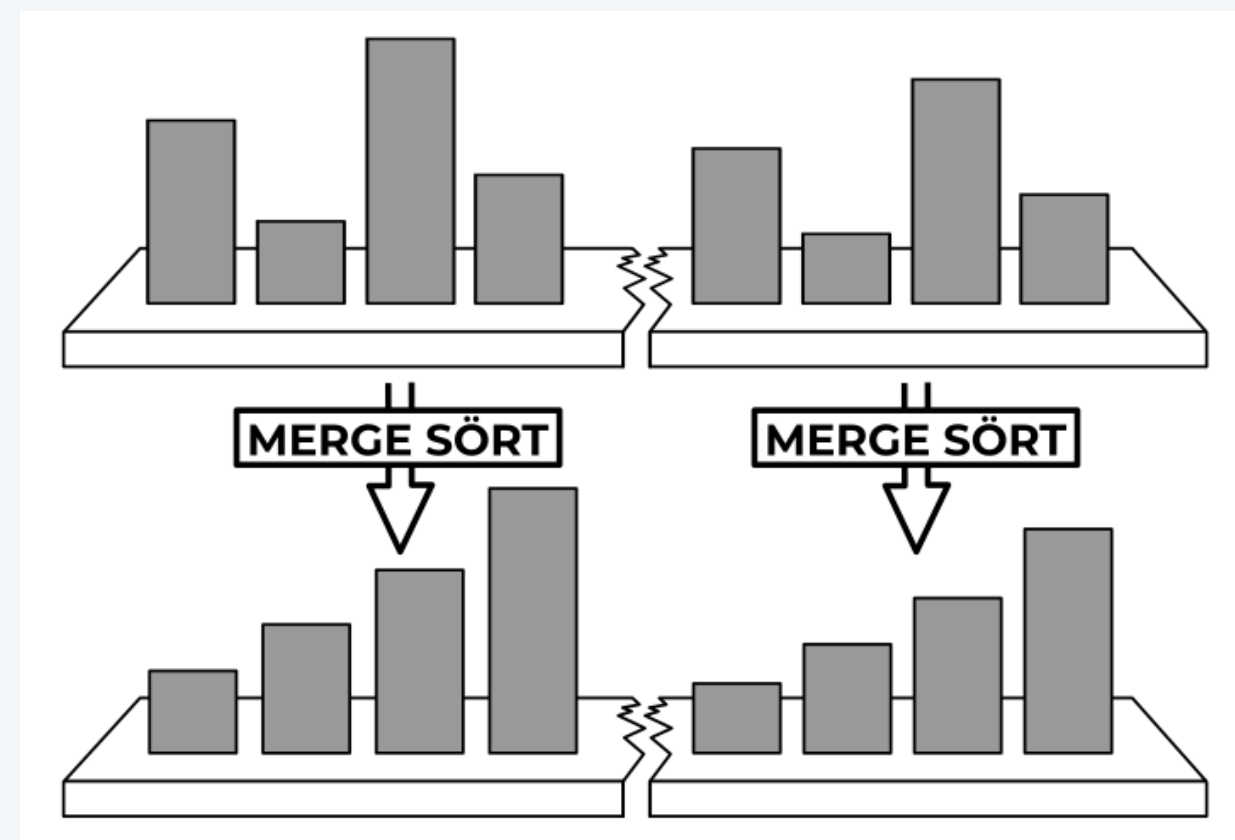
# Summary

---

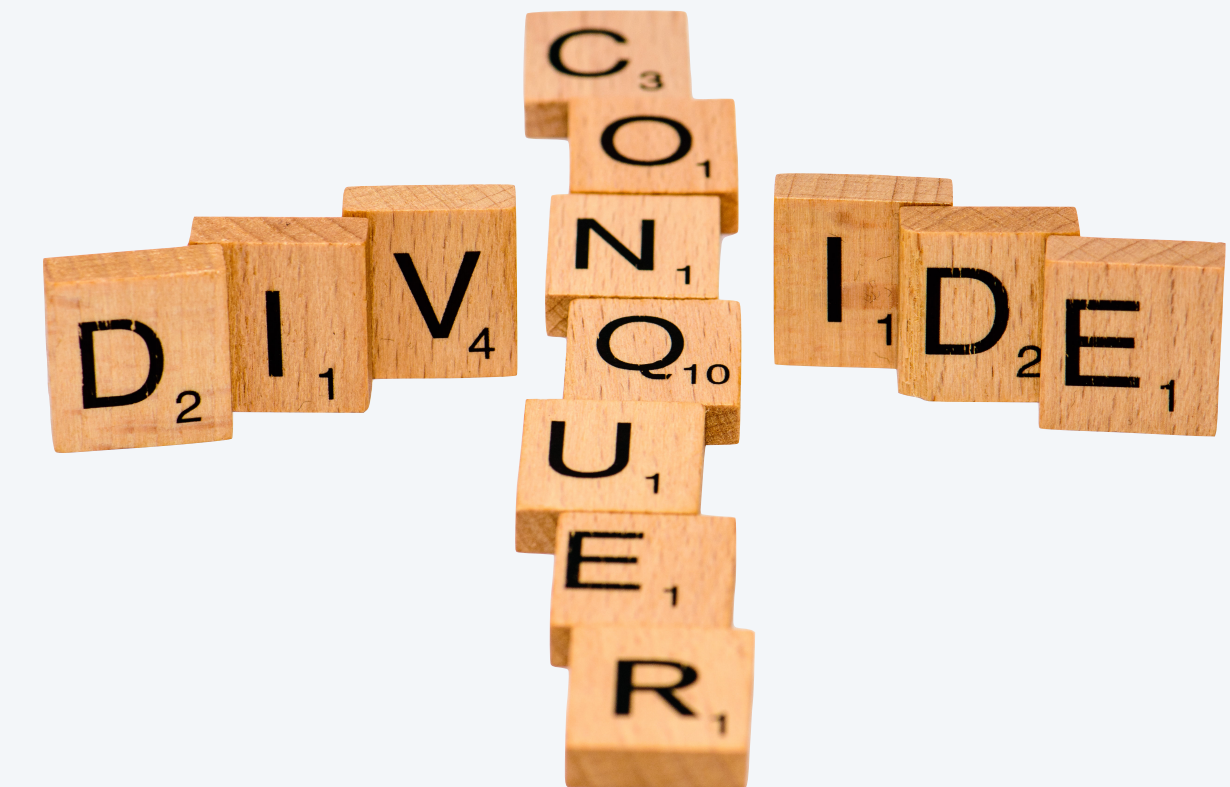
**Mergesort.** Makes  $O(n \log n)$  compares (and array accesses) in the worst case.

**Sorting lower bound.** In the worst case, any compare-based sorting algorithm makes  $\Omega(n \log n)$  compares.

**Divide-and-conquer.** Divide a problem into two (or more) subproblems; solve each subproblem independently; combine results.



**IMPOSSIBLE**



# Credits

---

media	source	license
<i>Jon von Neumann</i>	<u>IAS / Alan Richards</u>	
<i>Computer and Supercomputer</i>	<u>New York Times</u>	
<i>Mergesort Visualization</i>	<u>Toptal</u>	
<i>Tim Peters</i>	unknown	
<i>Flexing Arm</i>	<u>freepik.com</u>	
<i>Theory vs. Practice</i>	<u>Ela Sjolie</u>	
<i>Skittles Sorting Machine</i>	<u>Rolf R. Bakke</u>	
<i>Fast Skittles Sorting Machine</i>	<u>Kazumichi Moriyama</u>	
<i>Mergesort Instructions</i>	<u>IDEA</u>	<u>CC BY-NC-SA 4.0</u>
<i>Impossible Stamp</i>	<u>Adobe Stock</u>	<u>education license</u>
<i>Divide-and-Conquer Tiles</i>	<u>wallpapercrafter.com</u>	

# A final thought

## MERGE SÖRT

idea-instructions.com/merge-sort/  
v1.2, CC by-nc-sa 4.0

IDEA

