

# Efficient Sorting Algorithms

- ▶ mergesort
- ▶ sorting complexity
- ▶ quicksort
- ▶ animations

*Reference:*

*Algorithms in Java, Chapter 7, 8*

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

Except as otherwise noted, the content of this presentation is licensed under the Creative Commons Attribution 2.5 License.

## 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 20<sup>th</sup> century in science and engineering.

Mergesort.

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

Quicksort.

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

▶ mergesort

▶ sorting complexity

▶ quicksort

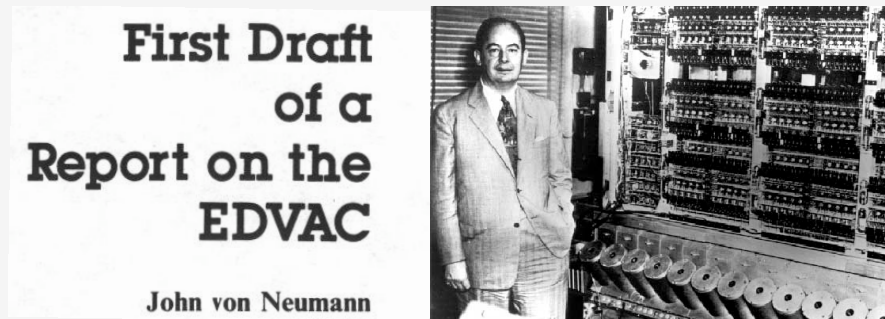
▶ animations

# Mergesort

## Basic plan.

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

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





# Merging

Goal. Combine two sorted subarrays into a sorted whole.

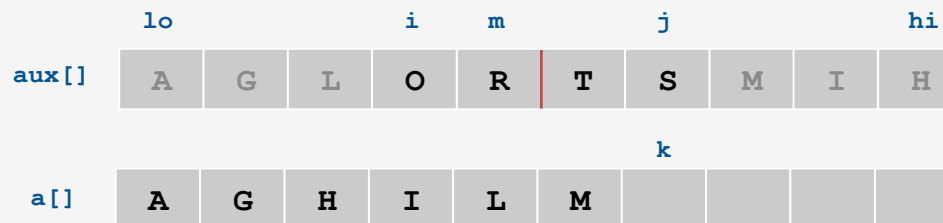
Q. How to merge efficiently?

A. Use an auxiliary array.

		a[]												aux[]											
		k	0	1	2	3	4	5	6	7	8	9	i	j	0	1	2	3	4	5	6	7	8	9	
<i>input</i>			E	E	G	M	R	A	C	E	R	T			-	-	-	-	-	-	-	-	-	-	-
<i>copy left</i>			E	E	G	M	R	A	C	E	R	T			E	E	G	M	R	-	-	-	-	-	-
<i>reverse copy right</i>			E	E	G	M	R	A	C	E	R	T			E	E	G	M	R	T	R	E	C	A	-
													0	9											
		0	A										0	8	E	E	G	M	R	T	R	E	C	A	
		1	A	C									0	7	E	E	G	M	R	T	R	E	C		
		2	A	C	E								1	7	E	E	G	M	R	T	R	E			
		3	A	C	E	E							2	7		E	G	M	R	T	R	E			
		4	A	C	E	E	E						2	6			G	M	R	T	R	E			
<i>merged result</i>		5	A	C	E	E	E	G					3	6			G	M	R	T	R				
		6	A	C	E	E	E	G	M				4	6				M	R	T	R				
		7	A	C	E	E	E	G	M	R			5	6					R	T	R				
		8	A	C	E	E	E	G	M	R	R		5	5						T	R				
		9	A	C	E	E	E	G	M	R	R	T	6	5							T				
<i>final result</i>			A	C	E	E	E	G	M	R	R	T													

## Merging: Java implementation

```
public static void merge(Comparable[] a, int lo, int m, int hi)
{
    for (int i = lo; i <= m; i++)
        aux[i] = a[i];
    for (int j = m+1; j <= hi; j++)
        aux[j] = a[hi-j+m+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++];
}
```



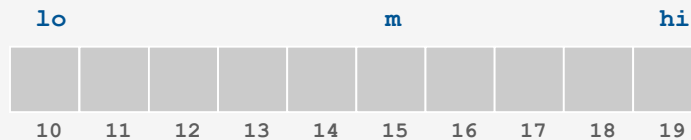
## Mergesort: Java implementation

```
public class Merge
{
    private static Comparable[] aux;

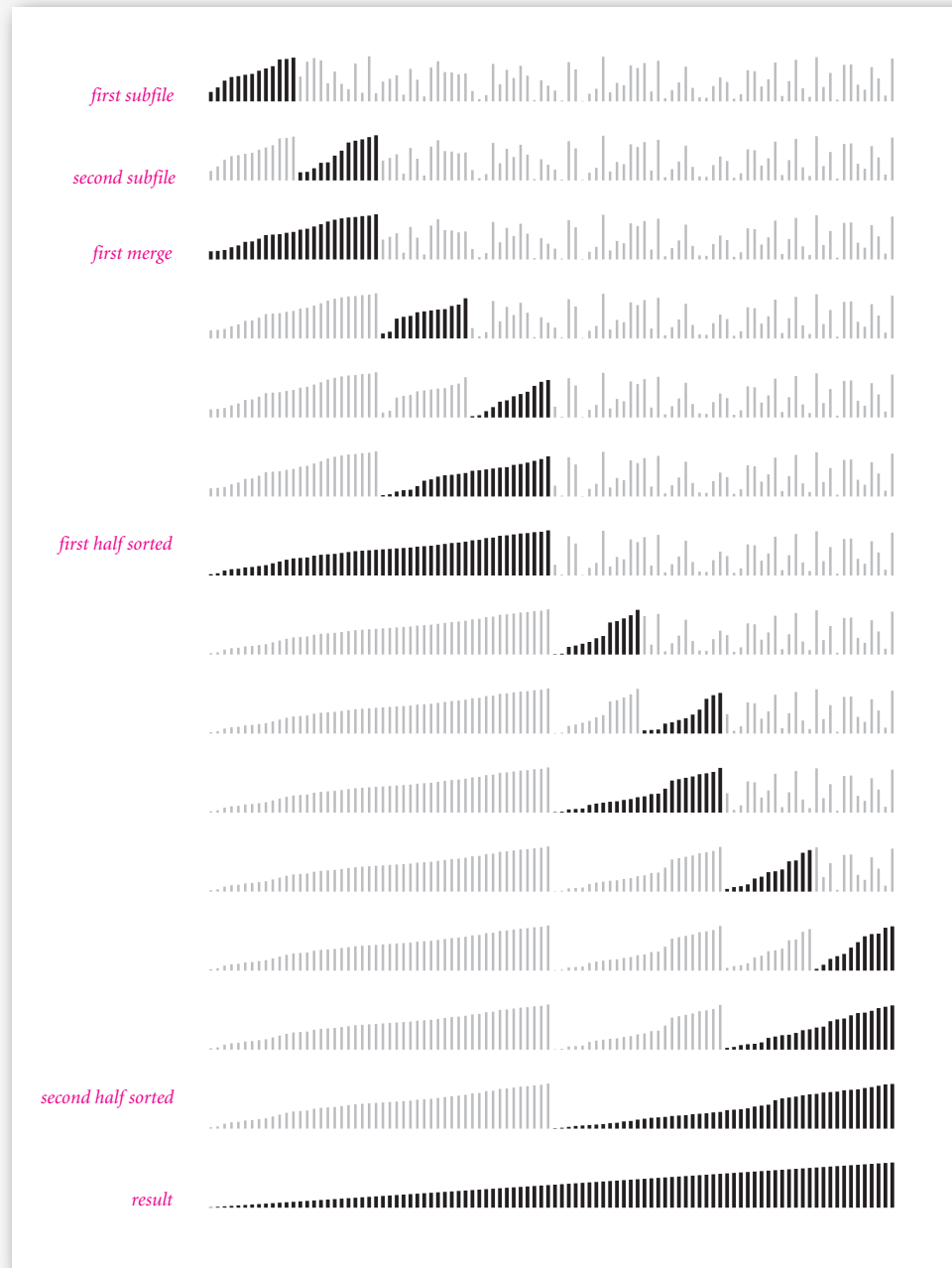
    private static void merge(Comparable[] a, int lo, int m, int hi)
    { /* as before */ }

    private static void sort(Comparable[] a, int lo, int hi)
    {
        if (hi <= lo) return;
        int m = lo + (hi - lo) / 2;
        sort(a, lo, m);
        sort(a, m+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 visualization



## 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


**Lesson.** Good algorithms are better than supercomputers.

## Mergesort: mathematical analysis

**Proposition.** Mergesort uses  $\sim N \lg N$  compares to sort any array of size  $N$ .

**Def.**  $T(N)$  = number of compares to mergesort an array of size  $N$ .

$$= T(N/2) + T(N/2) + N$$



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

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

**Solution.**  $T(N) \sim N \lg N$ .

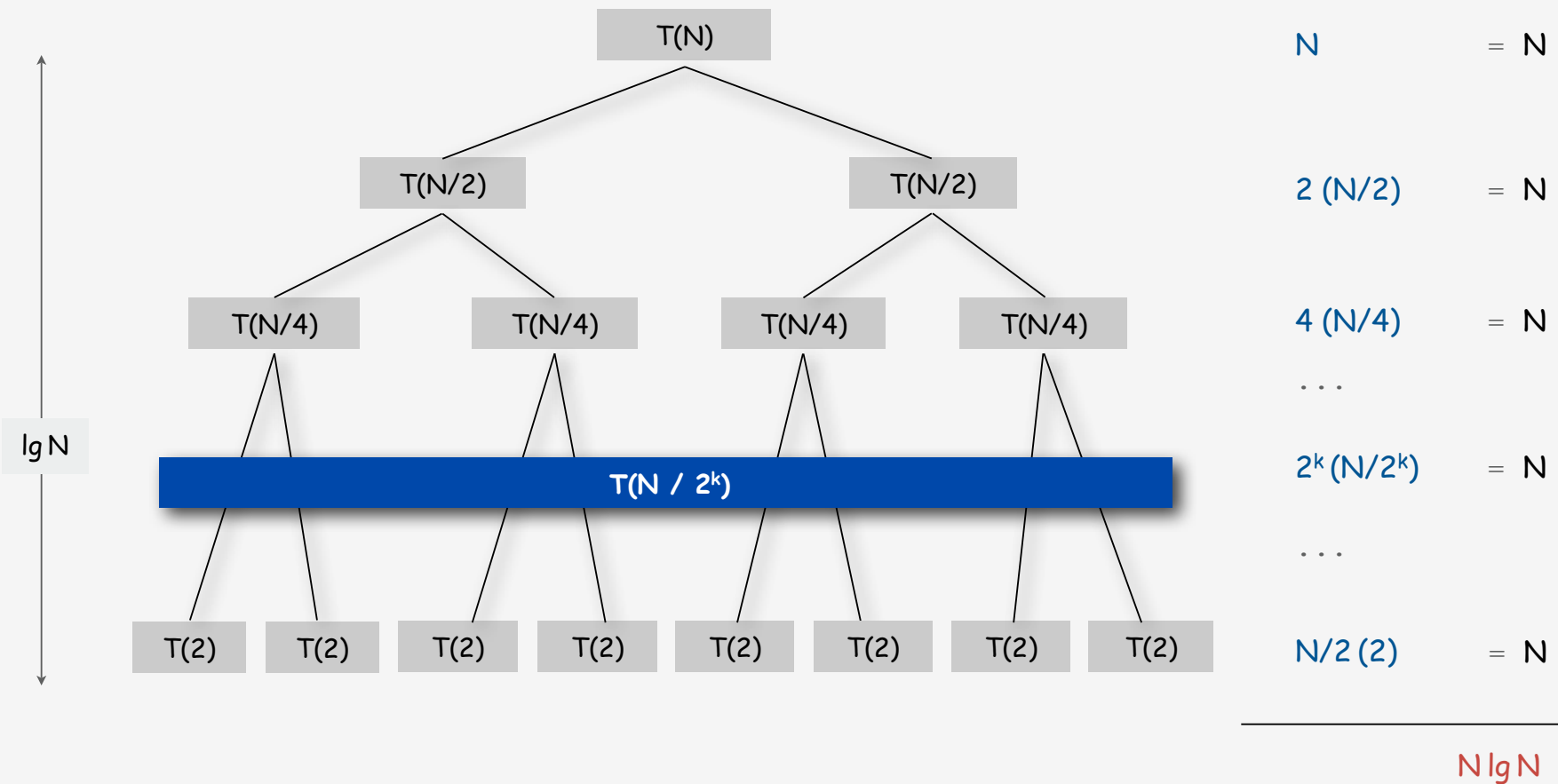
- For simplicity, we'll prove when  $N$  is a power of 2.
- True for all  $N$ . [see COS 340]

## Mergesort recurrence: proof 1

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

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

Pf.



## Mergesort recurrence: proof 2

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

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

Pf.

$$T(N) = 2 T(N/2) + N$$

$$T(N) / N = 2 T(N/2) / N + 1$$

$$= T(N/2) / (N/2) + 1$$

$$= T(N/4) / (N/4) + 1 + 1$$

$$= T(N/8) / (N/8) + 1 + 1 + 1$$

...

$$= T(N/N) / (N/N) + 1 + 1 + \dots + 1$$

$$= \lg N$$

given

divide both sides by  $N$

algebra

apply to first term

apply to first term again

stop applying,  $T(1) = 0$

## Mergesort recurrence: proof 3

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

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

**Pf.** [by induction on  $N$ ]

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

$$T(2N) = 2 T(N) + 2N$$

$$= 2 N \lg N + 2 N$$

$$= 2 N (\lg (2N) - 1) + 2N$$

$$= 2 N \lg (2N)$$

given

inductive hypothesis

algebra

QED

## 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.

<i>two sorted subarrays</i>	E	E	G	M	O	R	R	S		A	E	E	L	M	P	T	X
<i>merged array</i>	A	E	E	E	E	G	L	M	M	O	P	R	R	S	T	X	

**Def.** A sorting algorithm is **in-place** if it uses  $O(\log N)$  extra memory.

**Ex.** Insertion sort, selection sort, shellsort.

**Challenge for the bored.** In-place merge. [Kronrud, 1969]

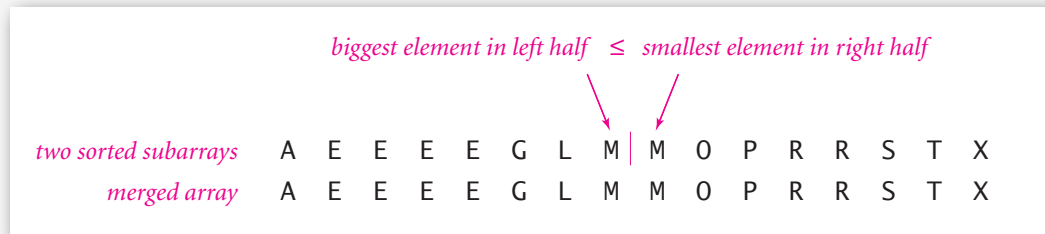
## Mergesort: practical improvements

Use insertion sort for small subarrays.

- Mergesort has too much overhead for tiny subarrays.
- Cutoff to insertion sort for  $\approx 7$  elements.

Stop if already sorted.

- Is biggest element in first half  $\leq$  smallest element in second half?
- Helps for nearly ordered lists.



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 Program 8.4 or `Arrays.sort()`.

## Bottom-up mergesort

### Basic plan.

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

	lo	m	hi	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
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

Bottom line. No recursion needed!

## Bottom-up mergesort: Java implementation

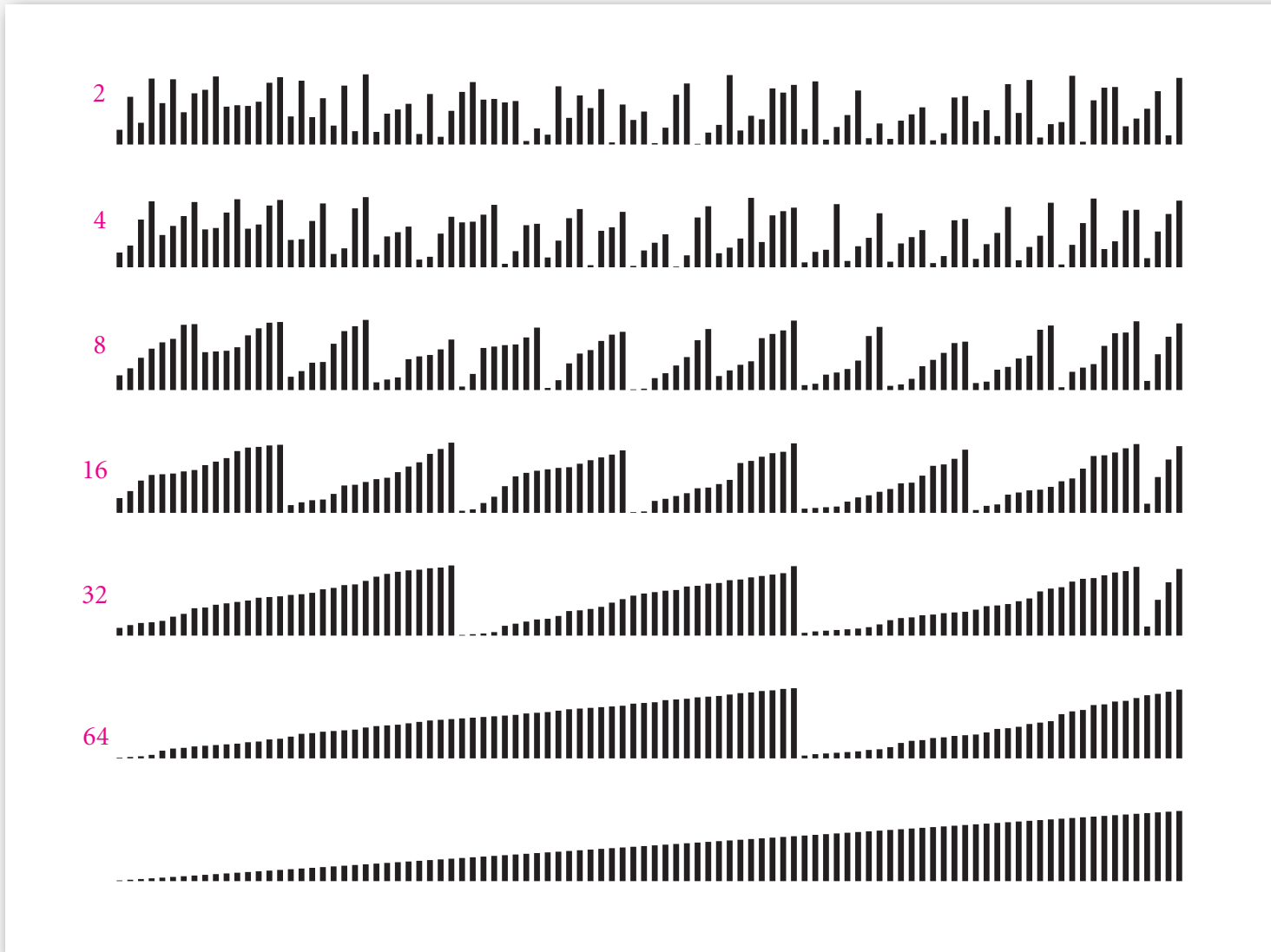
```
public class MergeBU
{
    private static Comparable[] aux;

    private static void merge(Comparable[] a, int lo, int m, int hi)
    { /* as before */ }

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

**Bottom line.** Concise industrial-strength code, if you have the space.

## Bottom-up mergesort: visualization



- ▶ mergesort
- ▶ **sorting complexity**
- ▶ quicksort
- ▶ animations

## 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 **any algorithm** for X.

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


lower bound ~ upper bound



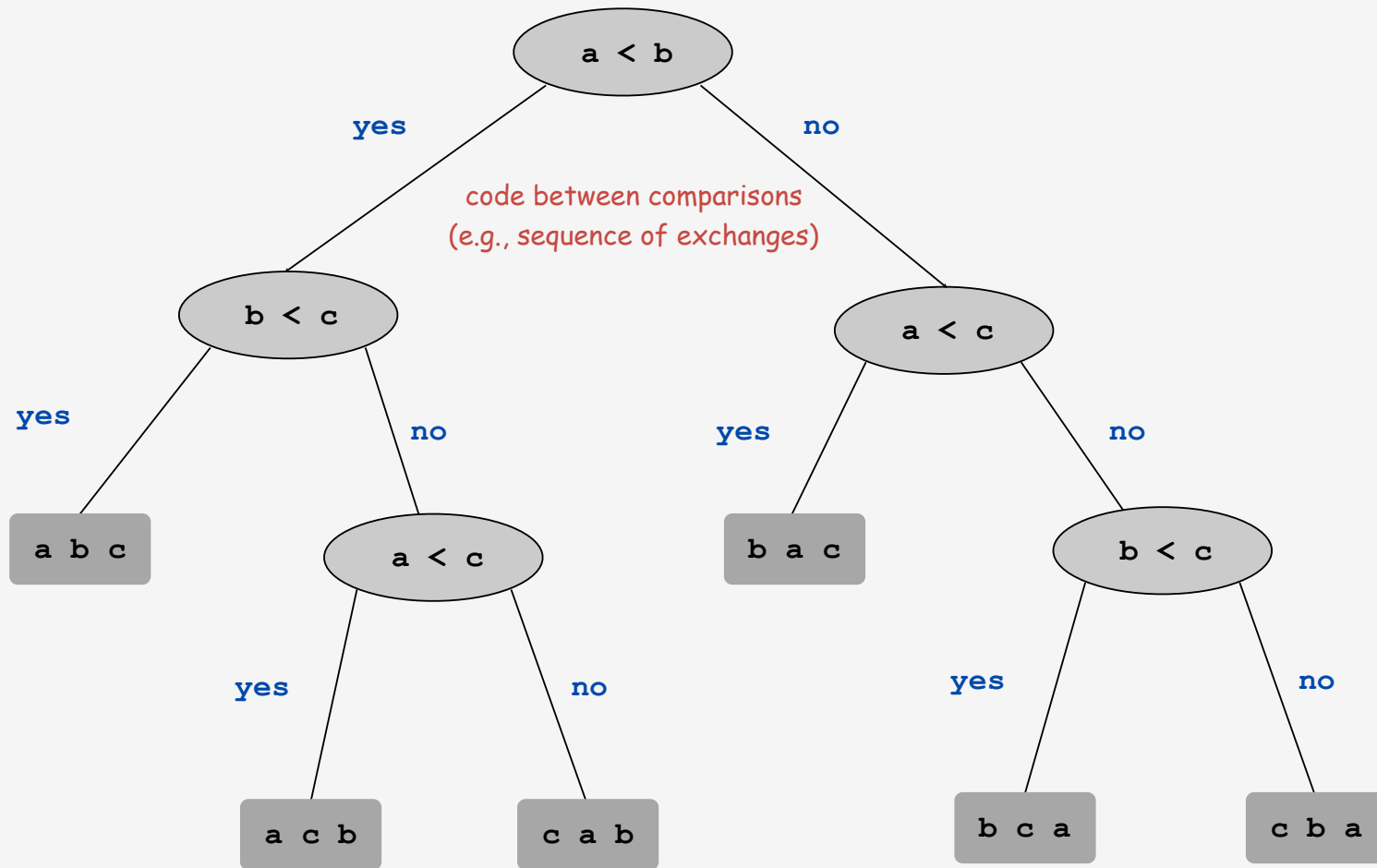
**Example: sorting.**

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

access information only through compares



# Decision tree

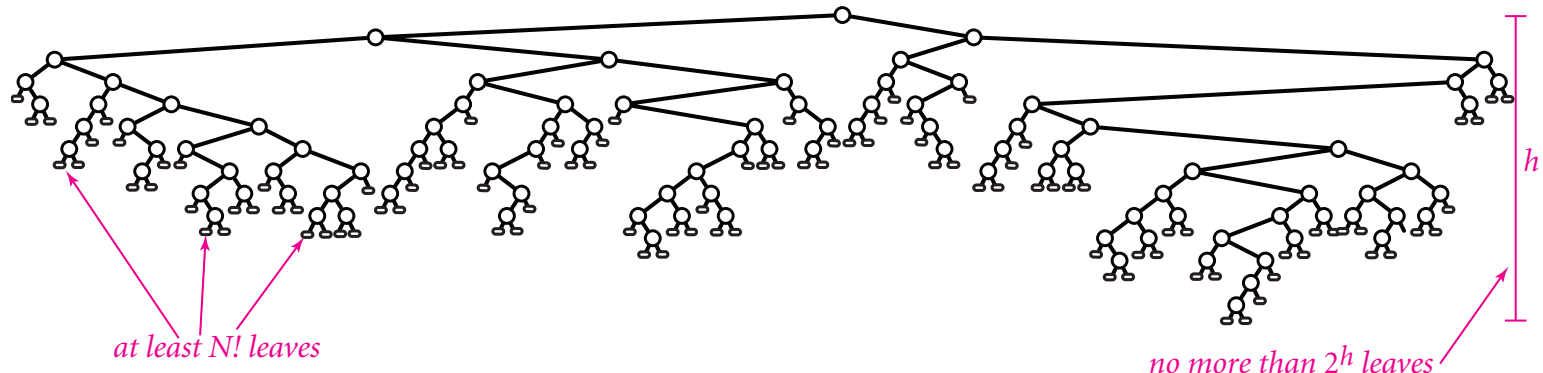


## Compare-based lower bound for sorting

**Proposition.** Any compare-based sorting algorithm must use more than  $N \lg N - 1.44 N$  comparisons 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 more than  $N \lg N - 1.44 N$  comparisons 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.

$$2^h \geq N!$$

$$h \geq \lg N!$$

$$\geq \lg(N/e)N \quad \leftarrow \text{Stirling's formula}$$

$$= N \lg N - N \lg e$$

$$\geq N \lg N - 1.44 N$$

## 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 any algorithm 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 =  $N \lg N - 1.44 N$ .
- 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.
- Is there 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 key values.
- Their initial arrangement.

**Partially ordered arrays.** Depending on the initial order of the input, we may not need  $N \lg N$  compares.

↖ insertion sort requires  $O(N)$  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 comparisons instead of key comparisons for numbers and strings.

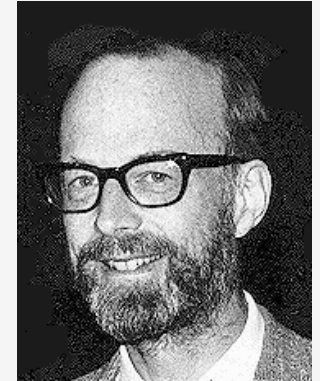
↖ stay tuned for radix sorts

- ▶ mergesort
- ▶ sorting complexity
- ▶ **quicksort**
- ▶ animations

# Quicksort

## Basic plan.

- **Shuffle** the array.
- **Partition** so that, for some  $i$ 
  - element  $a[i]$  is in place
  - no larger element to the left of  $i$
  - no smaller element to the right of  $i$
- **Sort** each piece recursively.



Sir Charles Antony Richard Hoare  
1980 Turing Award



## Quicksort partitioning

### Basic plan.

- Scan from left for an item that belongs on the right.
- Scan from right for item item that belongs on the left.
- Exchange.
- Continue until pointers cross.

	i	j	a[i]															
			0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
<i>initial values</i>	-1	15	E	R	A	T	E	S	L	P	U	I	M	Q	C	X	O	K
<i>scan left, scan right</i>	1	12	<u>E</u>	<u>R</u>	A	T	E	S	L	P	U	I	M	Q	<u>C</u>	<u>X</u>	<u>O</u>	K
<i>exchange</i>	1	12	E	C	A	T	E	S	L	P	U	I	M	Q	R	X	O	K
<i>scan left, scan right</i>	3	9	E	C	<u>A</u>	<u>T</u>	E	S	L	P	U	<u>I</u>	<u>M</u>	<u>Q</u>	R	X	O	K
<i>exchange</i>	3	9	E	C	A	I	E	S	L	P	U	T	M	Q	R	X	O	K
<i>scan left, scan right</i>	5	5	E	C	A	I	<u>E</u>	<u>S</u>	L	P	U	T	M	Q	R	X	O	K
<i>final exchange</i>	5	5	E	C	A	I	E	K	L	P	U	T	M	Q	R	X	O	S
<i>result</i>			E	C	A	I	E	K	L	P	U	T	M	Q	R	X	O	S

↙ v

*Partitioning trace (array contents before and after each exchange)*

## Quicksort: Java code for partitioning

```
private static int partition(Comparable[] a, int lo, int hi)
{
    int i = lo - 1;
    int j = hi;
    while(true)
    {
        while (less(a[++i], a[hi]))
            if (i == hi) break;

        while (less(a[hi], a[--j]))
            if (j == lo) break;

        if (i >= j) break;

        exch(a, i, j);

        exch(a, i, hi);
        return i;
    }
}
```

find item on left to swap

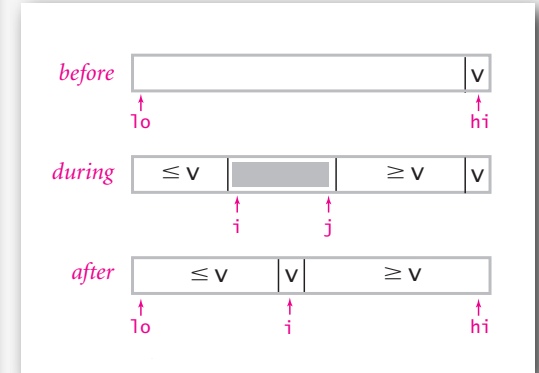
find item on right to swap

check if pointers cross

swap

swap with partitioning item

return index of item now known to be in place



## Quicksort: Java implementation

```
public class Quick
{
    public static void sort(Comparable[] a)
    {
        StdRandom.shuffle(a);
        sort(a, 0, a.length - 1);
    }

    private static void sort(Comparable[] a, int lo, int hi)
    {
        if (hi <= lo) return;
        int i = partition(a, lo, hi);
        sort(a, lo, i-1);
        sort(a, i+1, hi);
    }
}
```

# Quicksort trace

	lo	i	hi	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
<i>initial values</i>				Q	U	I	C	K	S	O	R	T	E	X	A	M	P	L	E
<i>random shuffle</i>				E	R	A	T	E	S	L	P	U	I	M	Q	C	X	O	K
	0	5	15	E	C	A	I	E	K	L	P	U	T	M	Q	R	X	O	S
	0	2	4	A	C	E	I	E	K	L	P	U	T	M	Q	R	X	O	S
	0	1	1	A	C	E	I	E	K	L	P	U	T	M	Q	R	X	O	S
	0		0	A	C	E	I	E	K	L	P	U	T	M	Q	R	X	O	S
	3	3	4	A	C	E	E	I	K	L	P	U	T	M	Q	R	X	O	S
	4		4	A	C	E	E	I	K	L	P	U	T	M	Q	R	X	O	S
	6	12	15	A	C	E	E	I	K	L	P	O	R	M	Q	S	X	U	T
	6	10	11	A	C	E	E	I	K	L	P	O	M	Q	R	S	X	U	T
	6	7	9	A	C	E	E	I	K	L	M	O	P	Q	R	S	X	U	T
	6		6	A	C	E	E	I	K	L	M	O	P	Q	R	S	X	U	T
	8	9	9	A	C	E	E	I	K	L	M	O	P	Q	R	S	X	U	T
	8		8	A	C	E	E	I	K	L	M	O	P	Q	R	S	X	U	T
	11		11	A	C	E	E	I	K	L	M	O	P	Q	R	S	X	U	T
	13	13	15	A	C	E	E	I	K	L	M	O	P	Q	R	S	T	U	X
	14	15	15	A	C	E	E	I	K	L	M	O	P	Q	R	S	T	U	X
	14		14	A	C	E	E	I	K	L	M	O	P	Q	R	S	T	U	X
<i>result</i>				A	C	E	E	I	K	L	M	O	P	Q	R	S	T	U	X

*initial values  
random shuffle*

*no partition  
for subarrays  
of size 1*

*result*

## Quicksort: implementation details

**Partitioning in-place.** Using a spare array makes partitioning easier, but is not worth the cost.

**Terminating the loop.** Testing whether the pointers cross is a bit trickier than it might seem.

**Staying in bounds.** The  $(i == hi)$  test is redundant, but the  $(j == lo)$  test is not.

**Preserving randomness.** Shuffling is key for performance guarantee.

**Equal keys.** When duplicates are present, it is (counter-intuitively) best to stop on elements equal to the partitioning element.

## Quicksort: empirical analysis

### Running time estimates:

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

computer	insertion sort ( $N^2$ )			mergesort ( $N \log N$ )			quicksort ( $N \log N$ )		
	thousand	million	billion	thousand	million	billion	thousand	million	billion
home	instant	2.8 hours	317 years	instant	1 second	18 min	instant	0.3 sec	6 min
super	instant	1 second	1 week	instant	instant	instant	instant	instant	instant

**Lesson 1.** Good algorithms are better than supercomputers.

**Lesson 2.** Great algorithms are better than good ones.

## Quicksort: average-case analysis

**Proposition I.** The average number of compares  $C_N$  to quicksort an array of  $N$  elements is  $\sim 2N \ln N$  (and the number of exchanges is  $\sim \frac{1}{3} N \ln N$ ).

**Pf.**  $C_N$  satisfies the recurrence  $C_0 = C_1 = 0$  and for  $N \geq 2$ :

$$C_N = \underbrace{(N+1)}_{\text{partitioning}} + \underbrace{(C_0 + C_1 + \dots + C_{N-1})}_{\text{left}} / N + \underbrace{(C_{N-1} + C_{N-2} + \dots + C_0)}_{\text{right}} / N$$

$\uparrow$   
partitioning probability

- Multiply both sides by  $N$  and collect terms:

$$NC_N = N(N+1) + 2(C_0 + C_1 + \dots + C_{N-1})$$

- Subtract this from the same equation for  $N-1$ :

$$NC_N - (N-1)C_N = 2N + 2C_{N-1}$$

- Rearrange terms and divide by  $N(N+1)$ :

$$C_N / (N+1) = (C_{N-1} / N) + 2 / (N+1)$$

## Quicksort: average-case analysis

- From before:

$$C_N / (N+1) = C_{N-1} / N + 2 / (N+1)$$

- Repeatedly apply above equation:

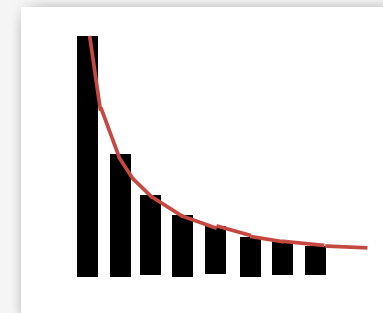
$$\begin{aligned} C_N / (N+1) &= C_{N-1} / N + 2 / (N+1) \\ &= C_{N-2} / (N-1) + 2/N + 2/(N+1) \\ &= C_{N-3} / (N-2) + 2/(N-1) + 2/N + 2/(N+1) \\ &= 2 ( 1 + 1/2 + 1/3 + \dots + 1/N + 1/(N+1) ) \end{aligned}$$

- Approximate by an integral:

$$\begin{aligned} C_N &\approx 2(N+1) ( 1 + 1/2 + 1/3 + \dots + 1/N ) \\ &= 2(N+1) H_N \approx 2(N+1) \int_1^N dx/x \end{aligned}$$

- Finally, the desired result:

$$C_N \approx 2(N+1) \ln N \approx 1.39 N \lg N$$



## Quicksort: summary of performance characteristics

**Worst case.** Number of compares is quadratic.

- $N + (N-1) + (N-2) + \dots + 1 \sim N^2 / 2$ .
- More likely that your computer is struck by lightning.

**Average case.** Number of compares is  $\sim 1.39 N \lg N$ .

- 39% more compares than mergesort.
- **But** faster than mergesort in practice because of less data movement.

**Random shuffle.**

- Probabilistic guarantee against worst case.
- Basis for math model that can be validated with experiments.

**Caveat emptor.** Many textbook implementations go **quadratic** if input:

- Is sorted or reverse sorted
- Has many duplicates (even if randomized!) [stay tuned]

## Quicksort: practical improvements

### Median of sample.

- Best choice of pivot element = median.
- Estimate true median by taking median of sample.

### Insertion sort small files.

- Even quicksort has too much overhead for tiny files.
- Can delay insertion sort until end.

### Optimize parameters.

- Median-of-3 random elements.
- Cutoff to insertion sort for  $\approx 10$  elements.

$\sim 12/7 N \lg N$  comparisons



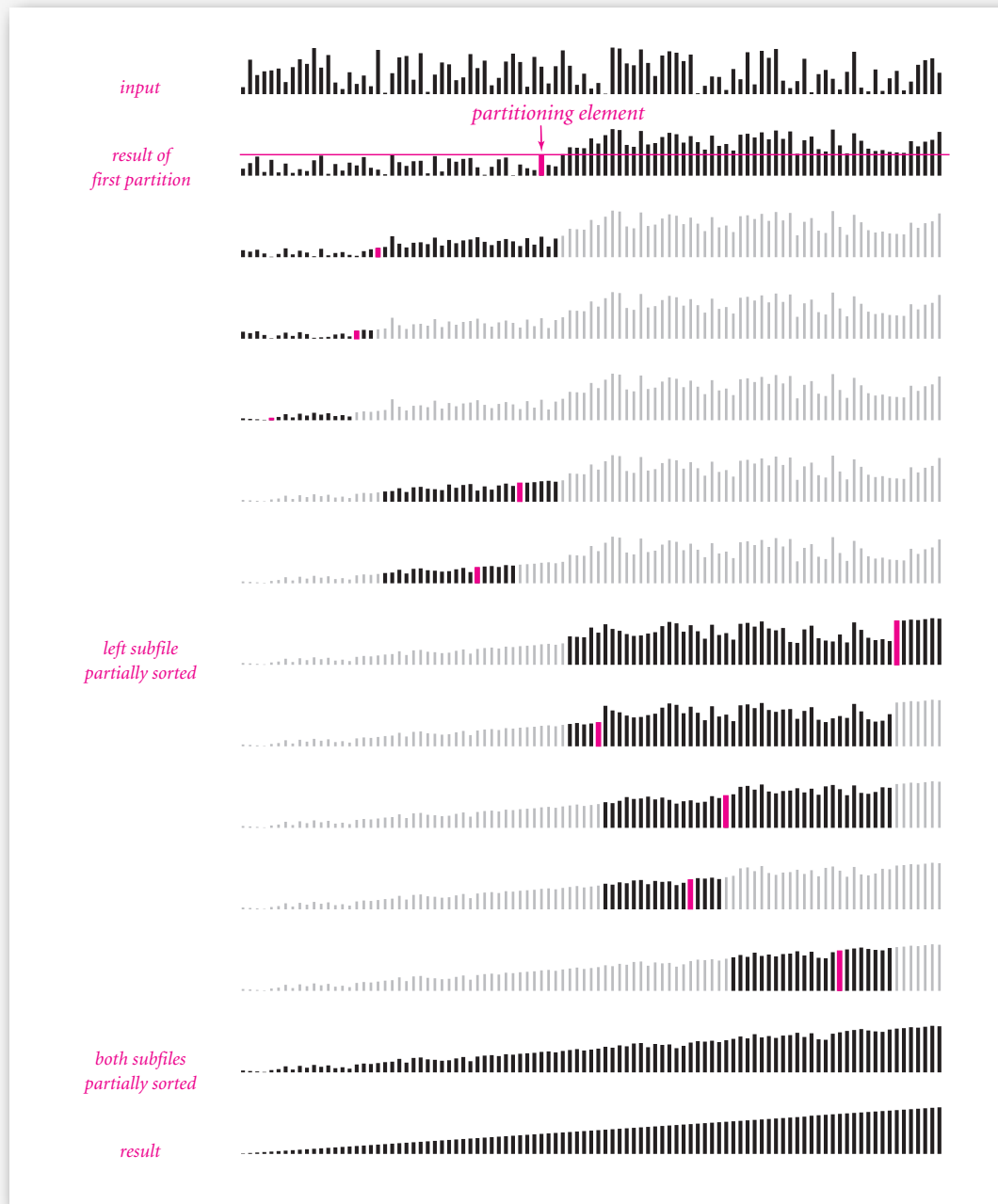
### Non-recursive version.

- Use explicit stack.
- Always sort smaller half first.

guarantees  $O(\log N)$  stack size

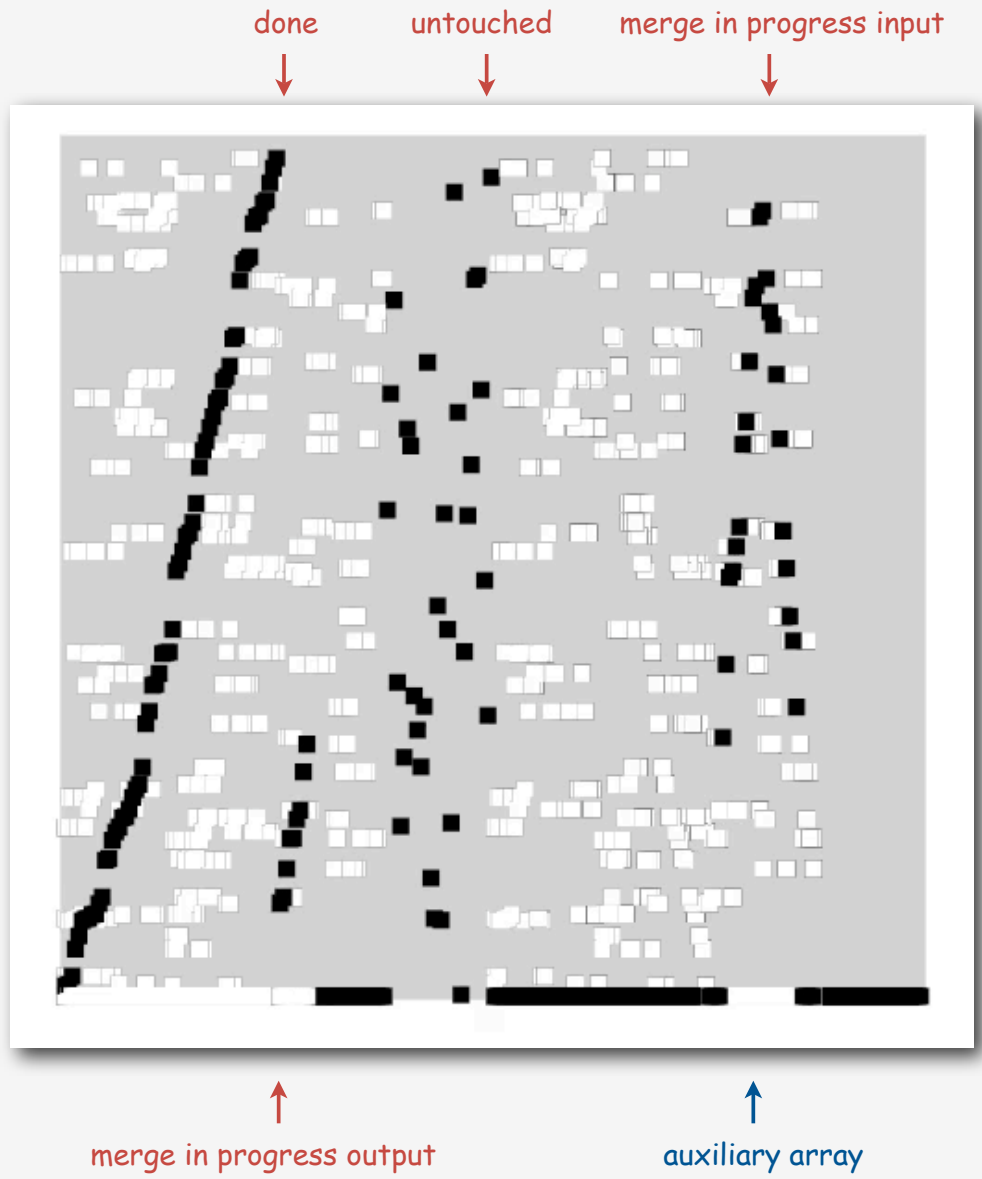


# Quicksort with cutoff to insertion sort: visualization

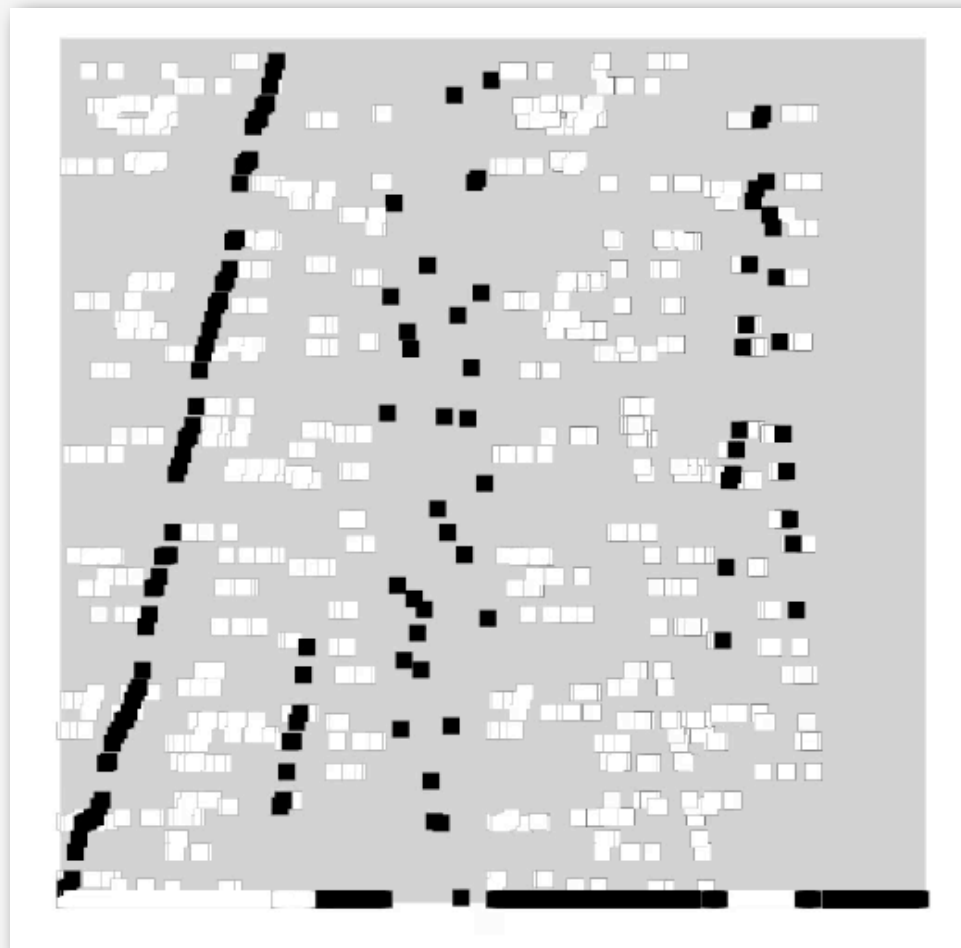


- ▶ mergesort
- ▶ sorting complexity
- ▶ quicksort
- ▶ **animations**

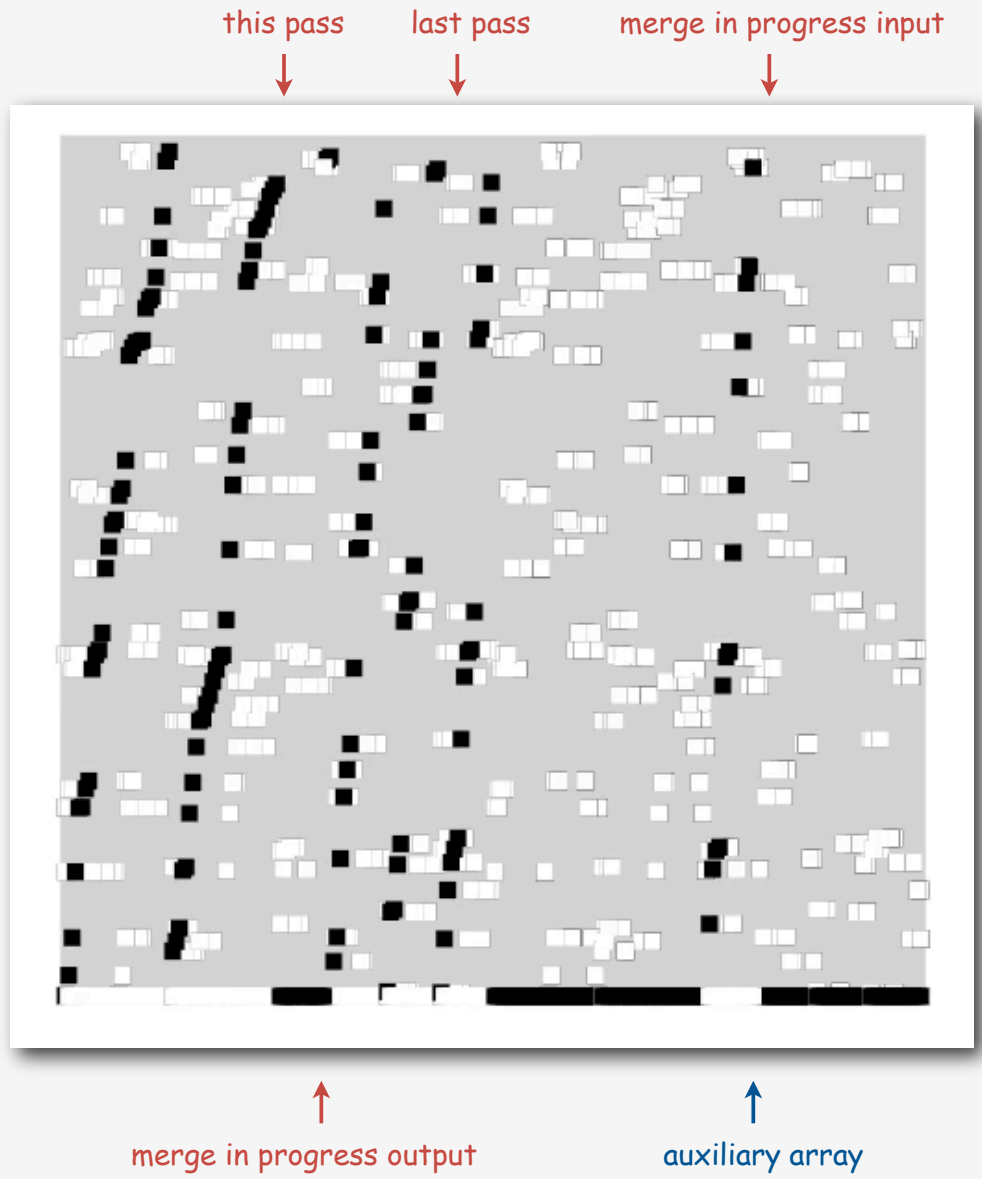
# Mergesort animation



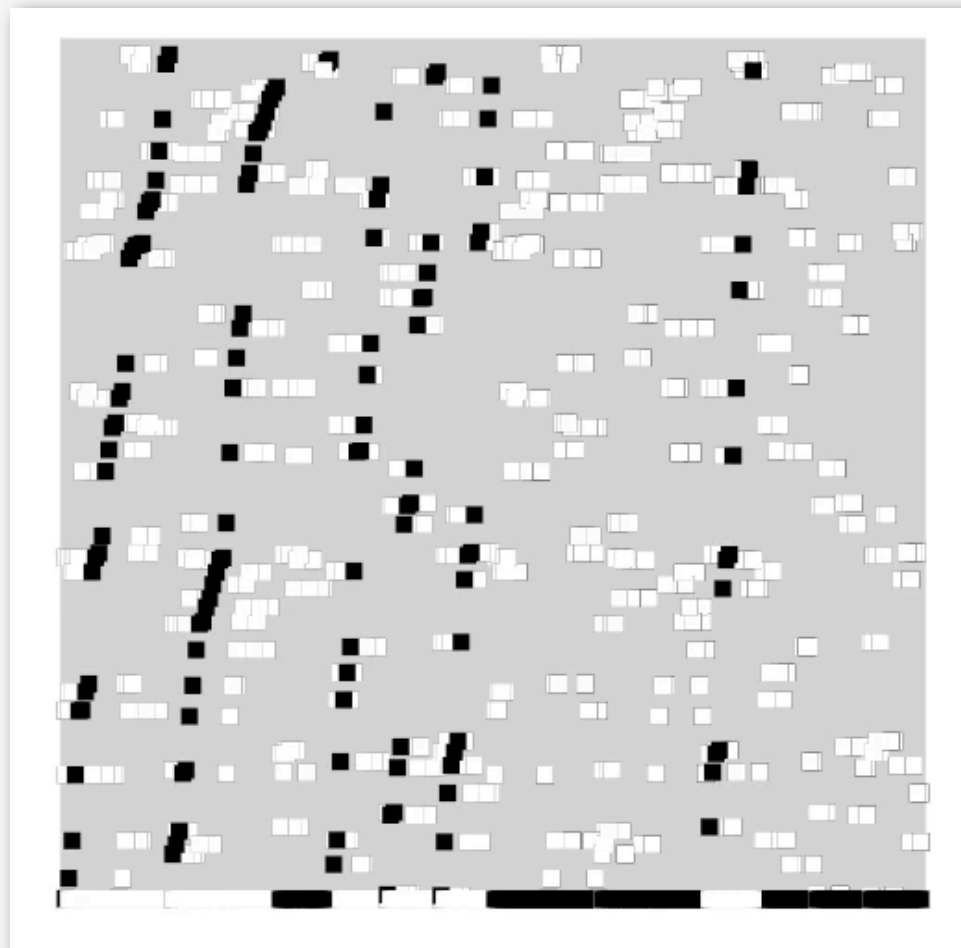
## Mergesort animation



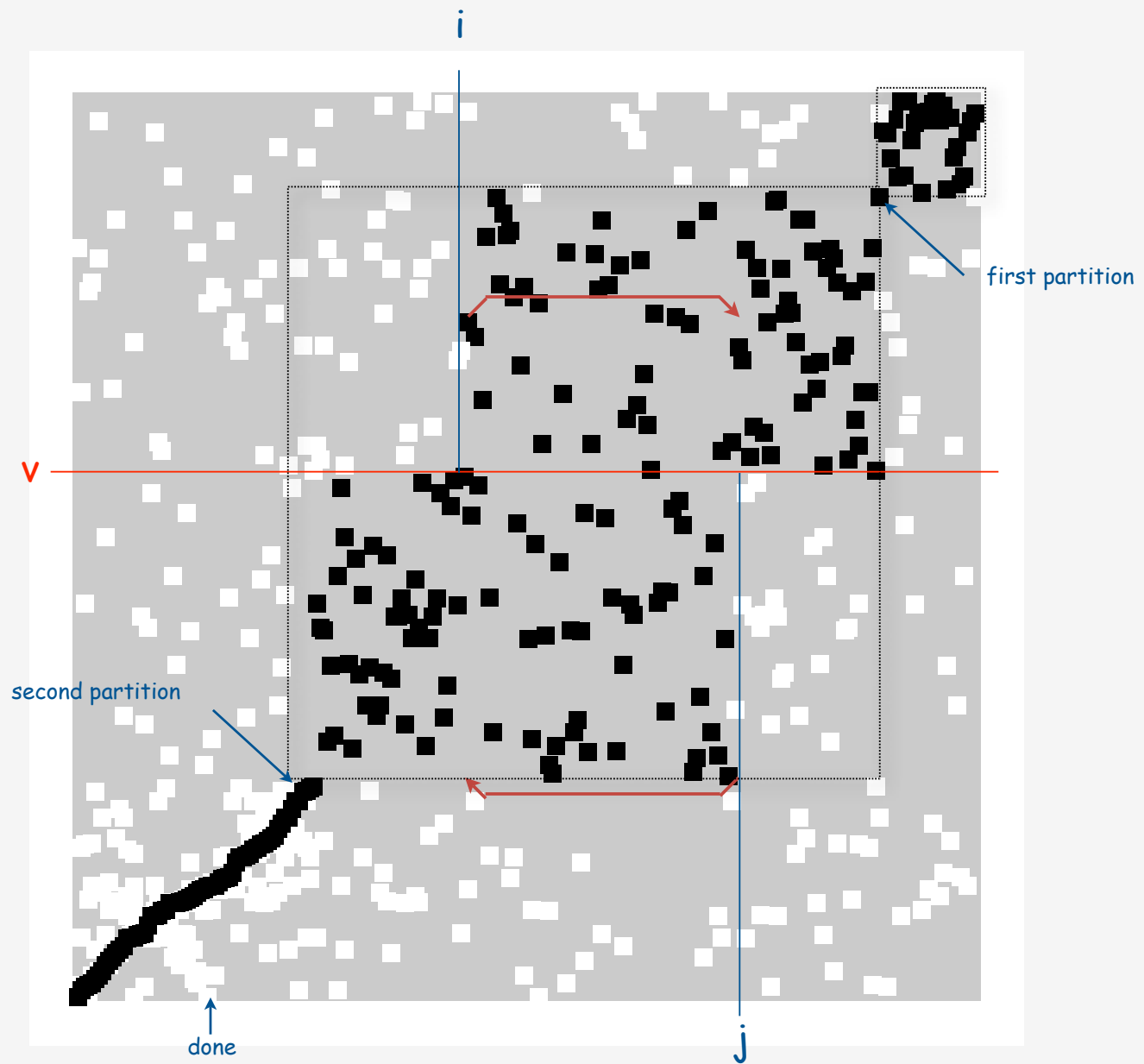
# Bottom-up mergesort animation



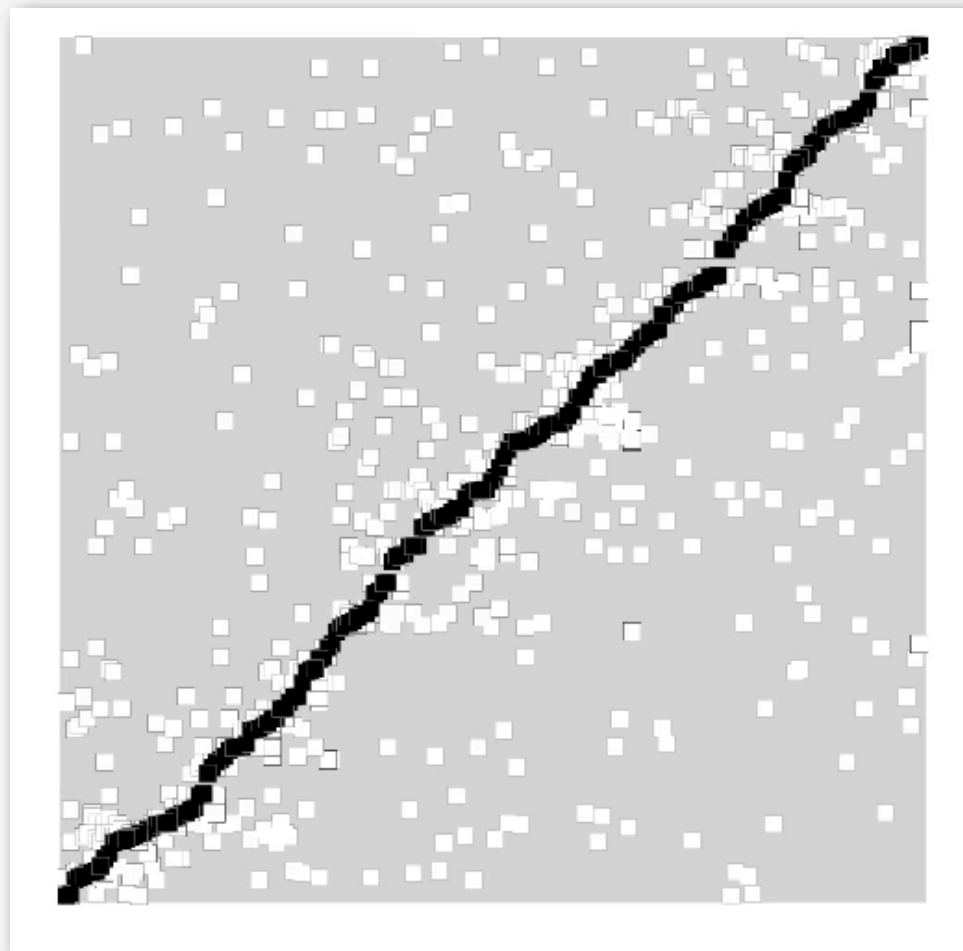
## Bottom-up mergesort animation



# Quicksort animation



## Quicksort animation



## Which sorting algorithm?

data	data	data	data	data	data	data	data
type	fifo	find	find	exch	hash	exch	exch
hash	hash	hash	hash	fifo	heap	fifo	fifo
heap	heap	heap	heap	find	type	heap	find
sort	exch	leaf	leaf	hash	link	find	hash
link	less	link	link	heap	list	link	heap
list	left	list	list	leaf	push	hash	leaf
push	leaf	push	push	left	sort	left	left
find	find	root	root	less	find	less	less
root	lifo	sort	sort	lifo	leaf	path	lifo
leaf	push	tree	tree	link	root	leaf	link
tree	tree	type	type	list	tree	lifo	list
null	null	exch	null	null	left	next	next
path	path	fifo	path	path	node	root	node
node	node	left	node	node	null	list	null
left	list	less	left	push	path	push	path
less	link	lifo	less	tree	exch	null	push
exch	sort	next	exch	type	less	swap	root
sink	sink	node	sink	sink	sink	node	sink
swim	swim	null	swim	swim	swim	swim	sort
next	next	path	next	next	fifo	sort	swap
swap	swap	sink	swap	swap	lifo	type	swim
fifo	type	swap	fifo	sort	next	sink	tree
lifo	root	swim	lifo	root	swap	tree	type
original	?	?	?	?	?	?	sorted

## Which sorting algorithm?

data	data	data	data	data	data	data	data
type	fifo	find	find	exch	hash	exch	exch
hash	hash	hash	hash	fifo	heap	fifo	fifo
heap	heap	heap	heap	find	type	heap	find
sort	exch	leaf	leaf	hash	link	find	hash
link	less	link	link	heap	list	link	heap
list	left	list	list	leaf	push	hash	leaf
push	leaf	push	push	left	sort	left	left
find	find	root	root	less	find	less	less
root	lifo	sort	sort	lifo	leaf	path	lifo
leaf	push	tree	tree	link	root	leaf	link
tree	tree	type	type	list	tree	lifo	list
null	null	exch	null	null	left	next	next
path	path	fifo	path	path	node	root	node
node	node	left	node	node	null	list	null
left	list	less	left	push	path	push	path
less	link	lifo	less	tree	exch	null	push
exch	sort	next	exch	type	less	swap	root
sink	sink	node	sink	sink	sink	node	sink
swim	swim	null	swim	swim	swim	swim	sort
next	next	path	next	next	fifo	sort	swap
swap	swap	sink	swap	swap	lifo	type	swim
fifo	type	swap	fifo	sort	next	sink	tree
lifo	root	swim	lifo	root	swap	tree	type
original	quicksort	mergesort	insertion	selection	merge BU	shellsort	sorted