SORTING

Jeff Erickson

SIGMA / SIAM

April 7, 2025

GINORST

---

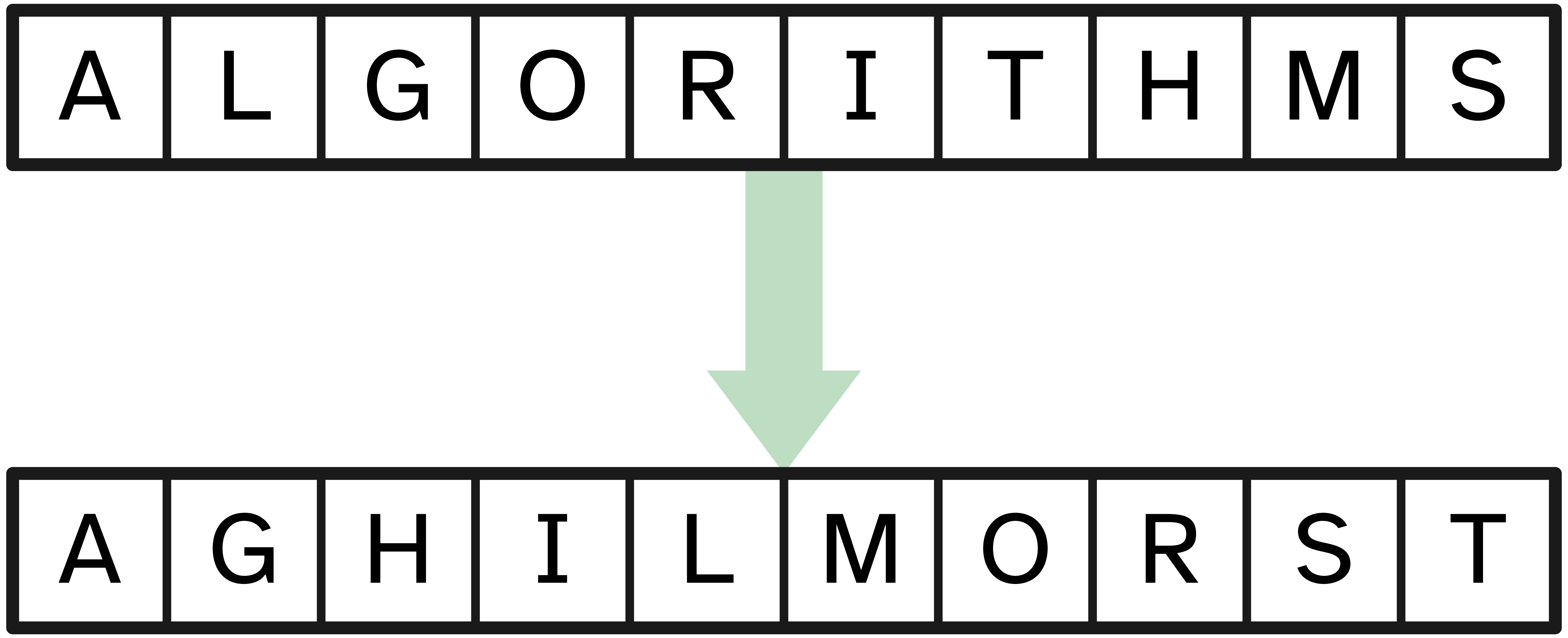
cEeffiJknors

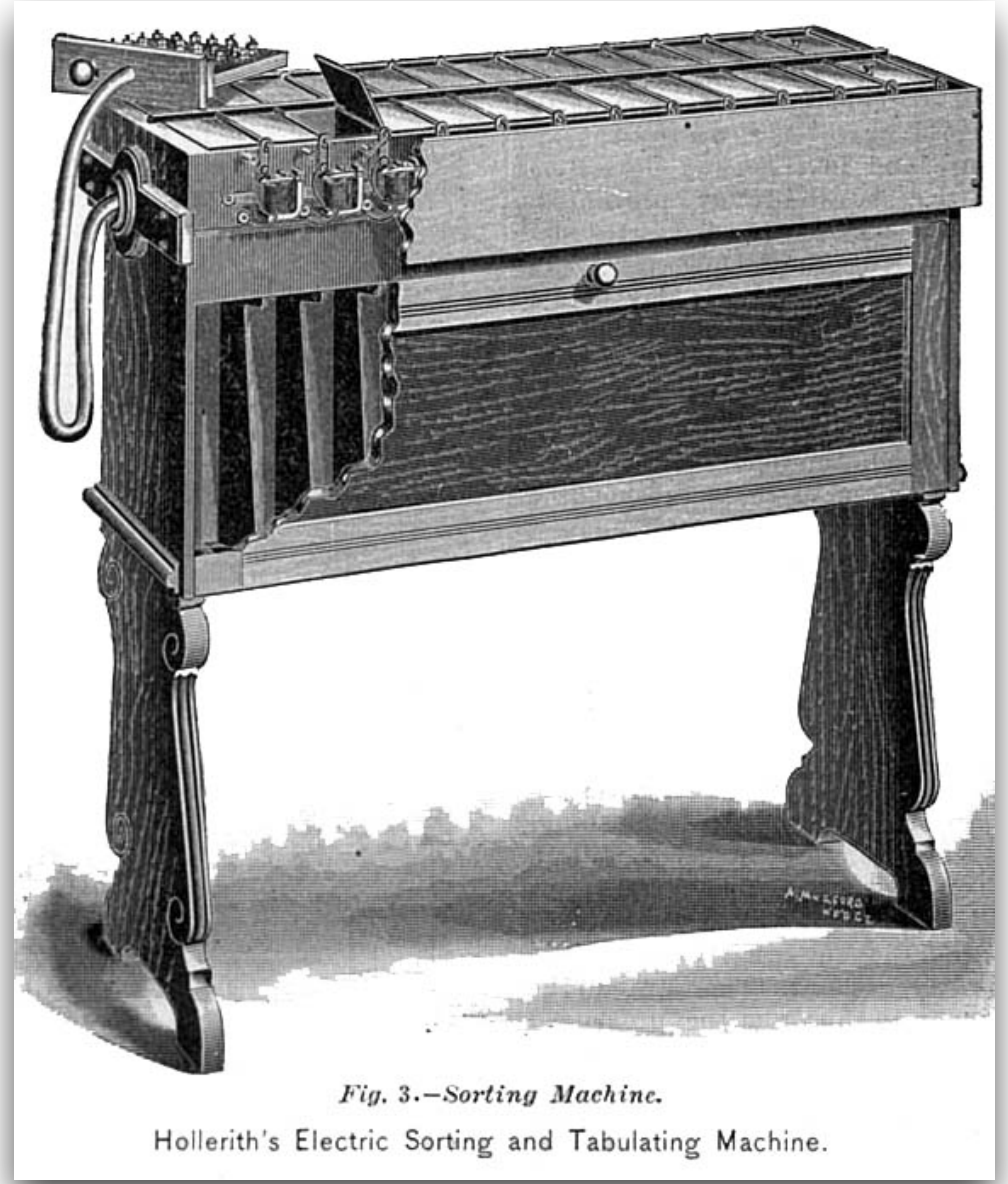
AAGIIMMSS/

Ø2257,Ailpr

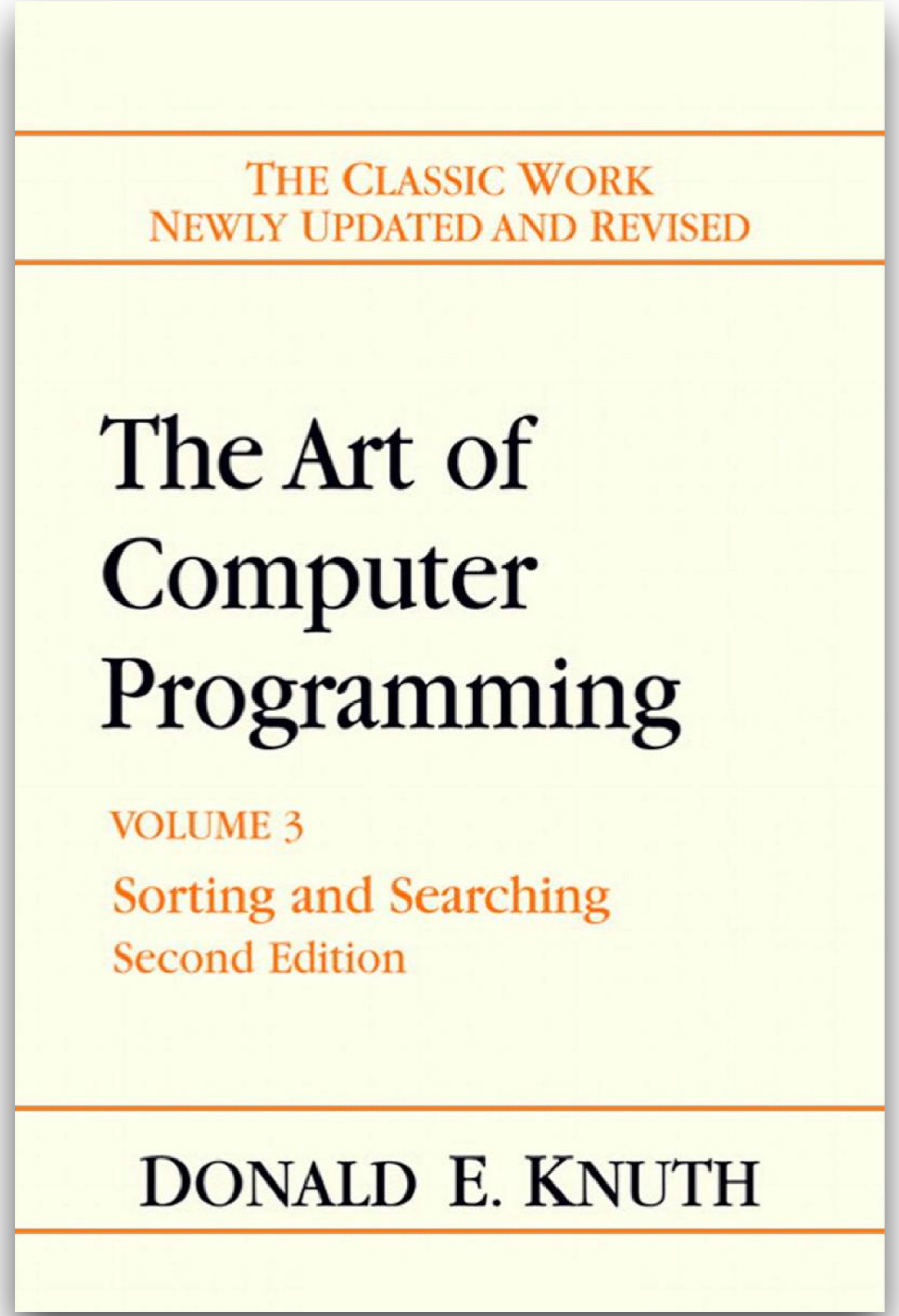
Sorting

- Given an array A[1…n], permute it so that A[i] < A[i+1] for all i.





[Hollerith 1890]



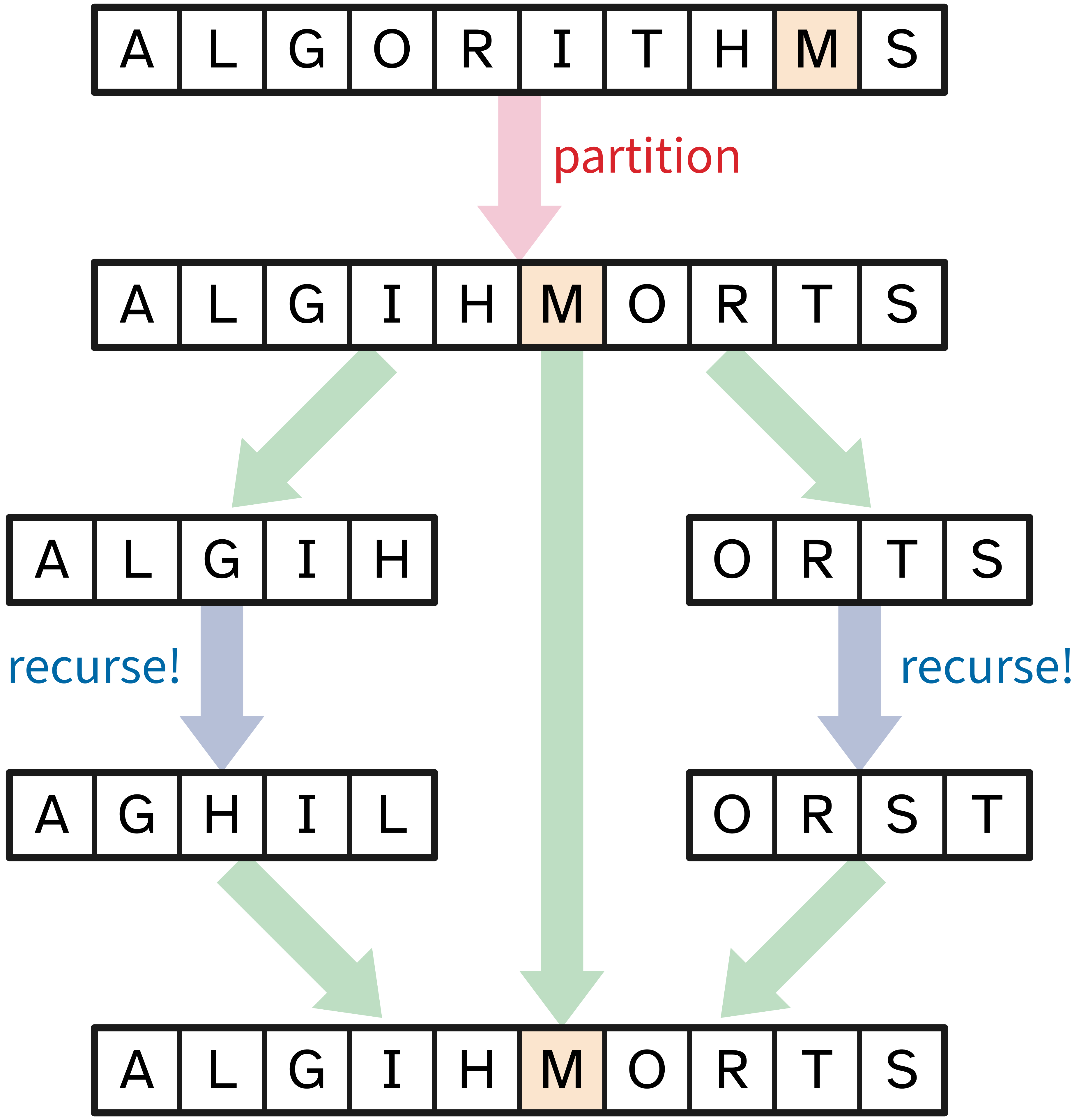
[Knuth 1973/1998]

Familiar sorting algorithms

- BubbleSort — $O(n^2)$ time
- SelectionSort — $O(n^2)$ time
- InsertionSort — $O(n^2)$ time
- MergeSort — $O(n \log n)$ time
- HeapSort — $O(n \log n)$ time
- QuickSort — It's complicated.

Quicksort

[Hoare 1959]

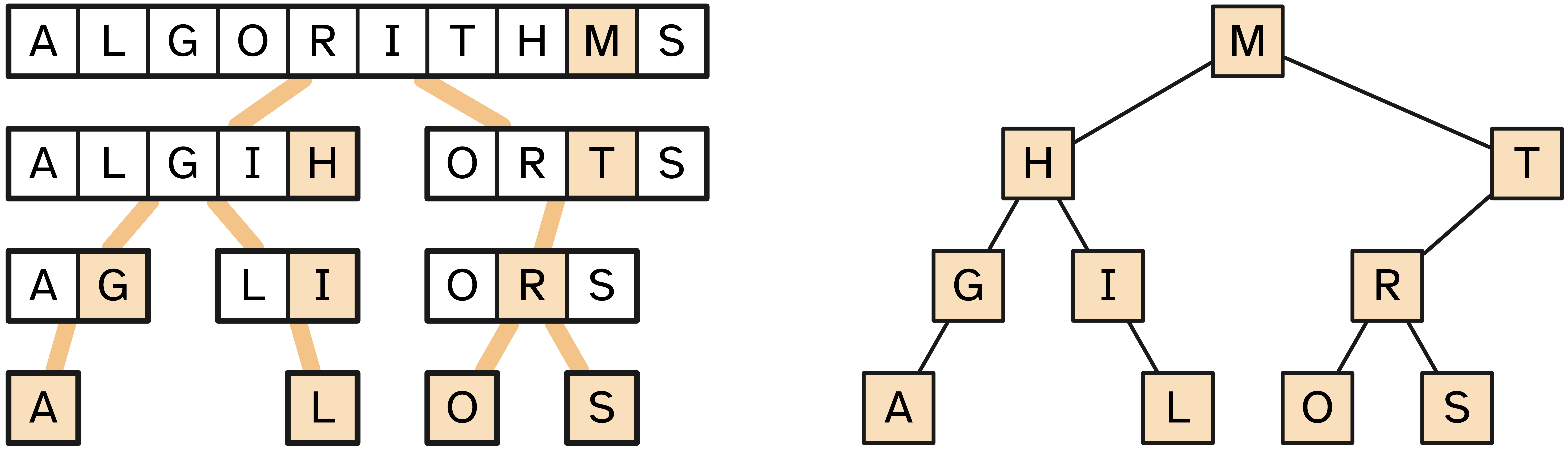


- Choose a pivot item p
- Partition array into items <p, =p, >p
- Recursively sort prefix and suffix

- Worst pivot: min or max ⇒ $\Theta(n^2)$ time
- Best pivot: median ⇒ $\Theta(n \log n)$ time

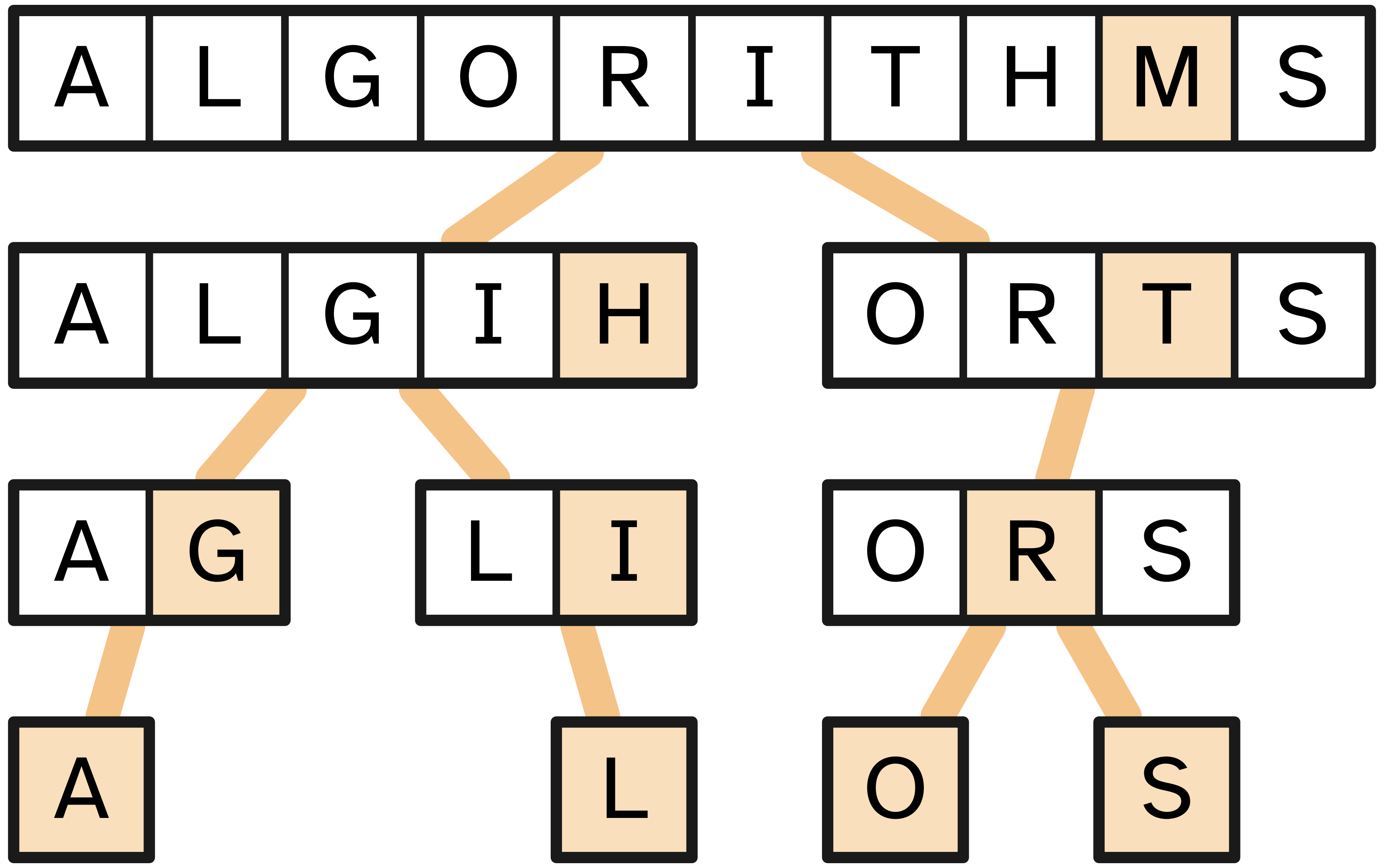
Quicksort recursion = BST

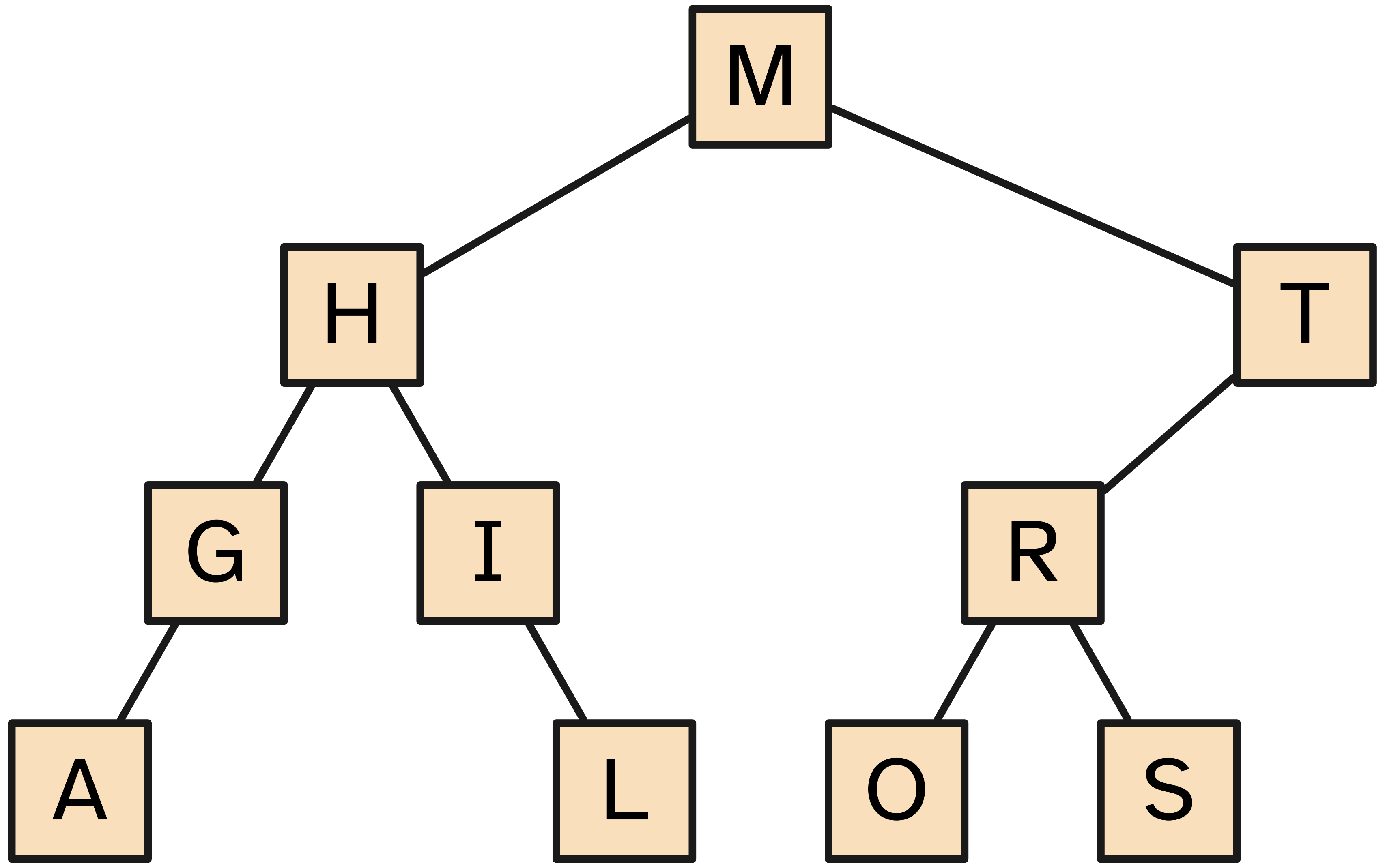
- Quicksort generates a binary tree of recursive calls.
- If we record the pivot at each node of this recursion tree, the result is a *binary search tree*!



Quicksort recursion = BST

- If we record the pivot at each node of this recursion tree, the result is a *binary search tree*!
- #comparisons = sum of node depths





Tree-insertion sort

```
TreeSort(A[1..n]):
  T ← new self-balancing binary tree
  for i ← 1 to n
      insert A[i] into T
  read A[1..n] from an in-order traversal of T
```

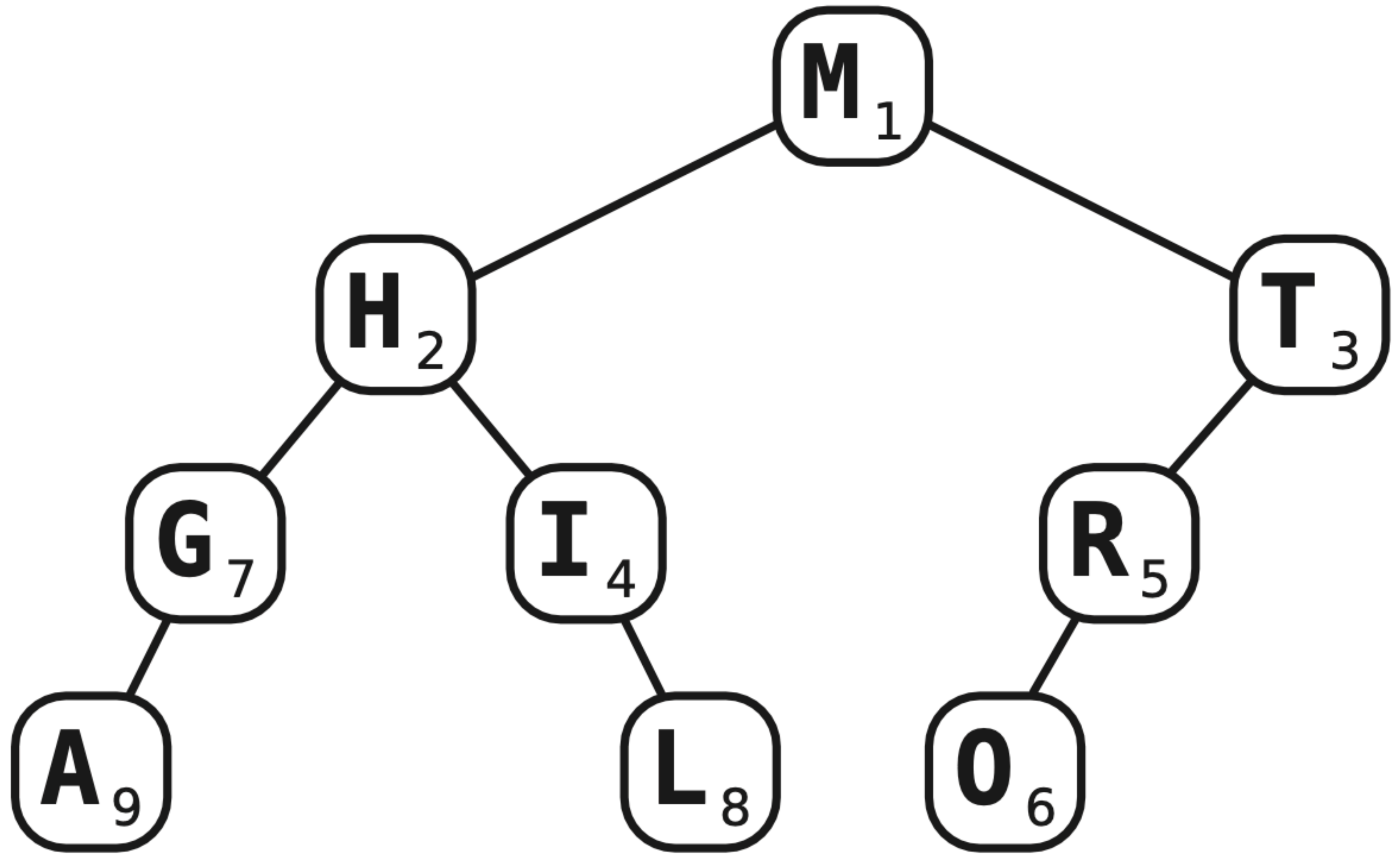
With **any** *self-balancing* binary search tree (AVL, red-black, splay, scapegoat, treaps, etc.), this algorithm runs in O(n log n) time.*

- **with high probability* for *randomized* BSTs (treaps, skip lists, etc.)

Treaps

[Seidel Aragon 1996]

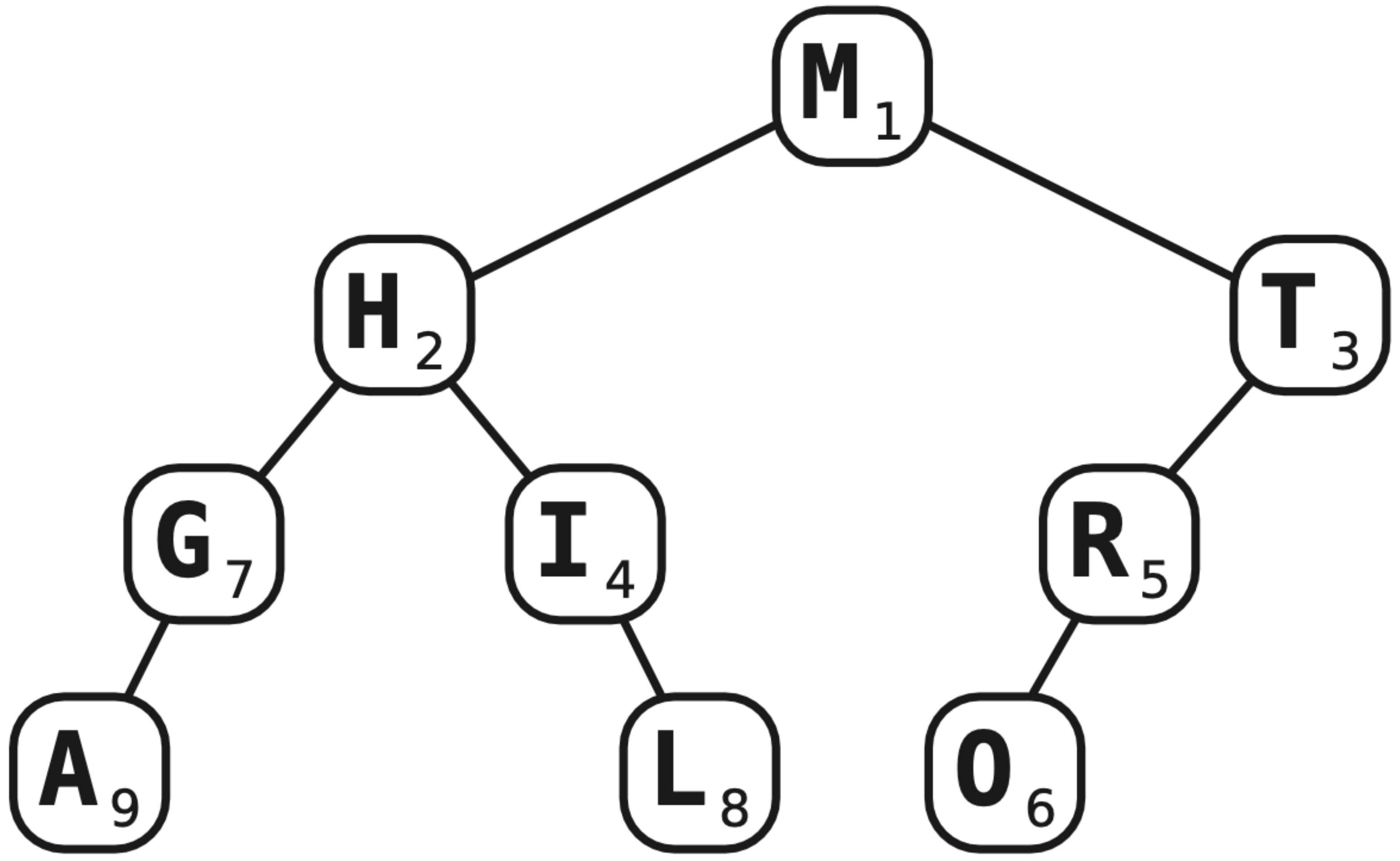
- Every node has a search key (given by the user) and a random priority (generated at insertion time).
- A treap is simultaneously a binary search tree for the search keys and a min-heap for the priorities.



Treaps

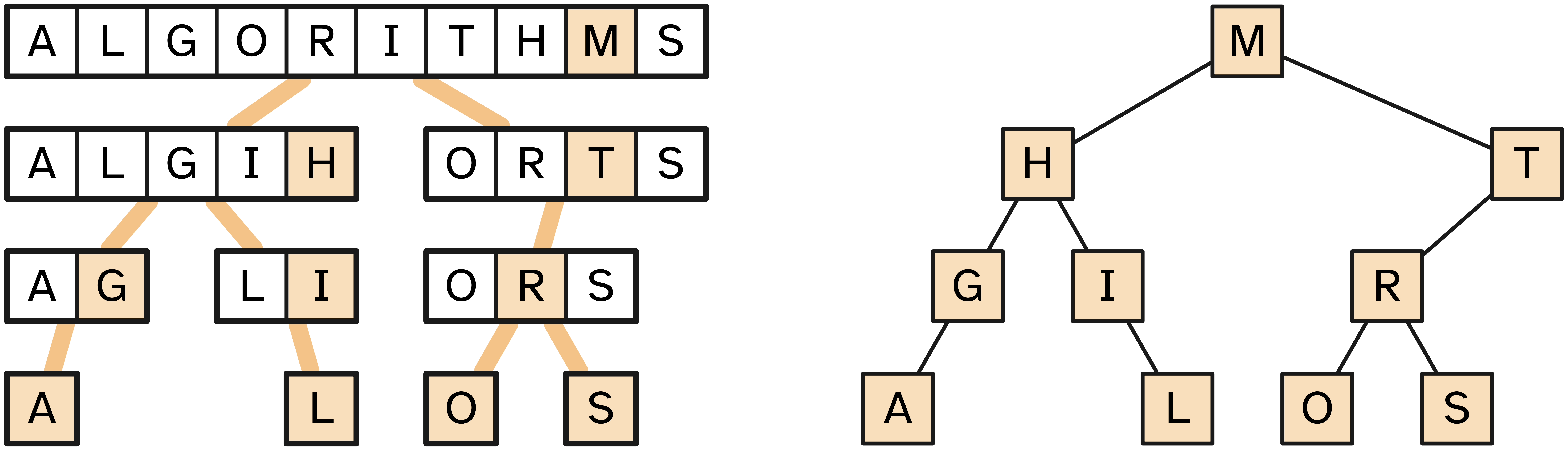
[Seidel Aragon 1996]

- Equivalently, insert keys into a standard binary search tree in increasing priority (= random) order.
- Random priorities guarantee depth O(log n) with high probability.



Randomized quicksort

- At each level of recursion, choose pivot uniformly at random
- The resulting recursion tree is a treap! So randomized quicksort runs in O(n log n) time with high probability!



Nuts and bolts

Nuts and bolts

[Rawlins 1992]

We wish to sort a bag of n nuts and n bolts by size in the dark. We can compare the sizes of a nut and a bolt by attempting to screw one into the other. This operation tells us that either the nut is bigger than the bolt; the bolt is bigger than the nut; or they are the same size (and so fit together). Because it is dark we are not allowed to compare nuts directly or bolts directly.
How many fitting operations do we need to sort the nuts and bolts in the worst case?

$O(n^2)$ steps is straightforward. (Hint: Find the largest bolt.)

Can we do better?

Nuts and bolts

Randomized quicksort! O(n log n) steps *with high probability*

- Choose a random pivot bolt
- Use pivot bolt to partition the nuts
- Use matching pivot nut to partition the bolts
- Recursively sort smaller nuts and bolts
- Recursively match larger nuts and bolts

Randomized mergesort: also O(n log n) *with high probability*

Deterministic nuts and bolts

- [Rawlins 1992]: problem first posed
- [Alon, Blum, Fiat, Kannan, Naor, Ostrovsky 1994]: $O(n \log^4 n)$
- [Bradford and Fleischer 1995]: $O(n \log^2 n)$
- [Bradford 1995] [Komlos Ma Szemerédi 1996]: $O(n \log n)$

However, all of these algorithms use *expander graphs*, which are easy to construct randomly, but hard to construct deterministically.

These results are best viewed as *noncontructive proofs* that fast deterministic algorithms exist!

Open question

Is there a deterministic algorithm that matches nuts and bolts in $O(n \log n)$ time?

Vibesort

Vibesort

[Fung 2021]

```
VibeSort(A[1..n]):
 for i ← 1 to n
     for j ← 1 to n
         if A[i] < A[j]
             swap A[i] ↔ A[j]
```

This is *obviously* wrong.

Vibesort

[Fung 2021]

```
VibeSort(A[1..n]):
  for i ← 1 to n
    for j ← 1 to n
      if A[i] < A[j]
        swap A[i] ↔ A[j]
```

```
SelectionSort(A[1..n]):
  for i ← 1 to n
    for j ← i+1 to n
      if A[i] > A[j]
        swap A[i] ↔ A[j]
```

```
InsertionSort(A[1..n]):
  for i ← 1 to n
    for j ← 1 to i–1
      if A[i] < A[j]
        swap A[i] ↔ A[j]
```

Vibesort

[Fung 2021]

```
VibeSort(A[1..n]):
  for i ← 1 to n
      for j ← 1 to n
          if A[i] < A[j]
              swap A[i] ↔ A[j]
```

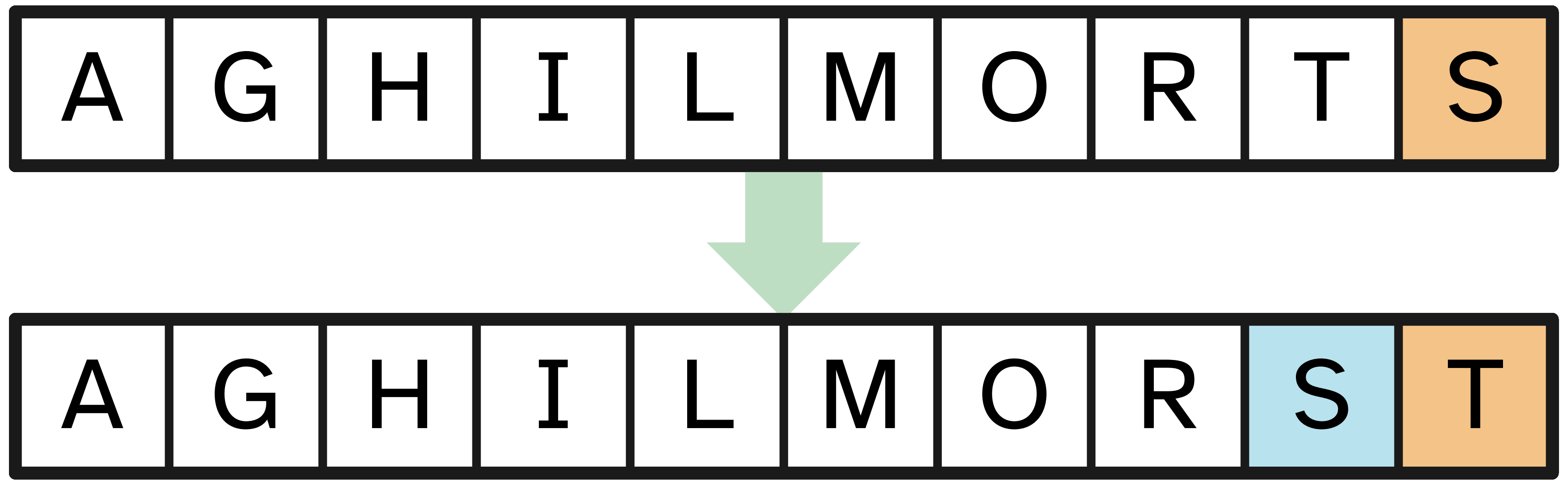
This is *obviously* wrong.

This is *actually* correct!

Vibesort

[Fung 2021]

```
VibeSort(A[1..n]):
  for i ← 1 to n
    for j ← 1 to n
      if A[i] < A[j]
        swap A[i] ↔ A[j]
```



- After i iterations, A[i] is the largest item in the array
- During ith iteration (except i=1) the suffix A[i+1..n] does not change.
- After i iterations of the outer loop, the prefix A[1..i] is sorted.
- *It's just insertion sort!*

“Vibesort”

[Fung 2021]

There is nothing good about this algorithm. It is slow – the algorithm obviously runs in $\Theta(n^2)$ time, whether worst-case, average-case or best-case. It unnecessarily compares all pairs of positions, twice (but see Section 3). There seems to be no intuition behind it, and its correctness is not entirely obvious. You certainly do not want to use it as a first example to introduce students to sorting algorithms. It is not stable, does not work well for external sorting, cannot sort inputs arriving online, and does not benefit from partially sorted inputs. Its only appeal may be its simplicity, in terms of lines of code and the “symmetry” of the two loops.

It is difficult to imagine that this algorithm was not discovered before, but we are unable to find any references to it.

Harmonic exchange

Random exchange

[Olesker-Taylor 2025]

```
RandExSort(A[1 .. n]):
  for k ← 1 to N
      choose random indices i < j
      if A[i] > A[j]
          swap A[i] ↔ A[j]
```

For *large enough* N, this algorithm sorts *with high probability,* where "*large enough*" depends on the probability distribution of pairs (i, j).

Random exchange

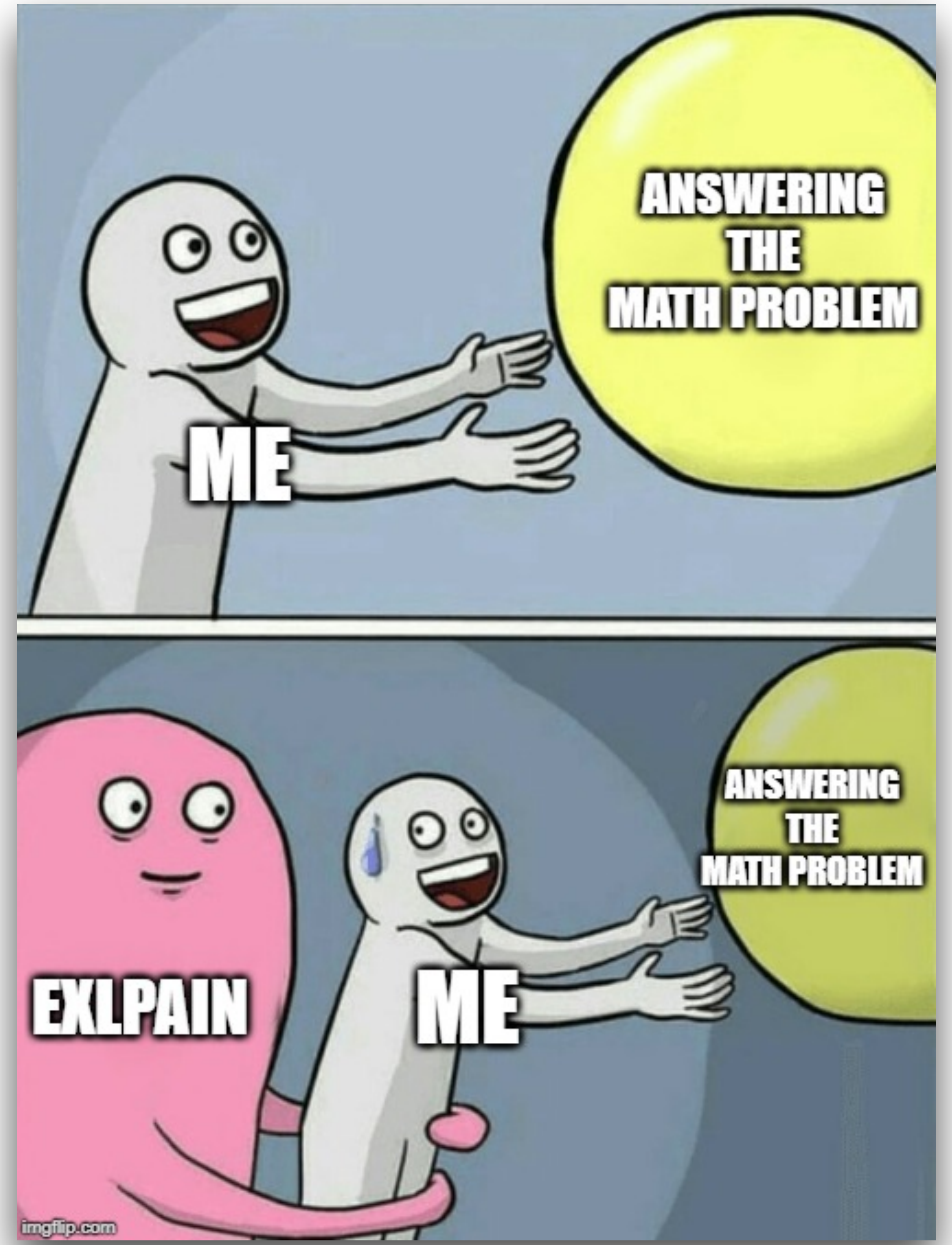
[Olesker-Taylor 2025]

```
RandExSort(A[1 .. n]):
  for k ← 1 to N
      choose random indices i < j
      if A[i] > A[j]
         swap A[i] ↔ A[j]
```

Uniform:	$\Pr[i, j] = 2/n(n-1)$	$N = \Theta(n^2 \log n)$
Adjacent:	$\Pr[i, i+1] = 1/(n-1)$	$N = \Theta(n^2)$
Harmonic:	$\Pr[i, j] \propto 1/(j-i)$	$N = \Theta(n \log^2 n)$

Harmonic intuition

[Olesker-Taylor 2025]



Random exchange intuition [Olesker-Taylor 2025]

- **Uniform:** In each iteration, each item is moved *directly* to its correct position with probability $\Theta(1/n^2)$. Other moves don't help.
- **Adjacent:** Each iteration moves two items *at most one step* closer to their correct positions; total distance could be $\Theta(n^2)$.
- **Harmonic:** "On average, each iteration moves two items *roughly halfway* to their correct positions."

Harmonic intuition

[Olesker-Taylor 2025]

We actually need Pr[i, j] ~ 1/(j–i)(n ln n) so that probabilities sum to 1.

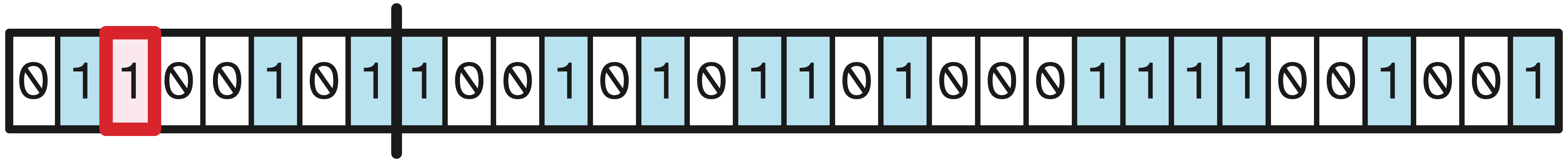
$$\sum_{i<j} \frac{1}{j-i} \sim nH_{n-1} \sim n \ln n$$

This is the same analysis as randomized quicksort!

Harmonic intuition

[Olesker-Taylor 2025]

- Suppose the input array A[1..n] contains exactly n/2 0s and n/2 1s.
- Consider a single “bad” 1 in the bottom quarter A[1...n/4].

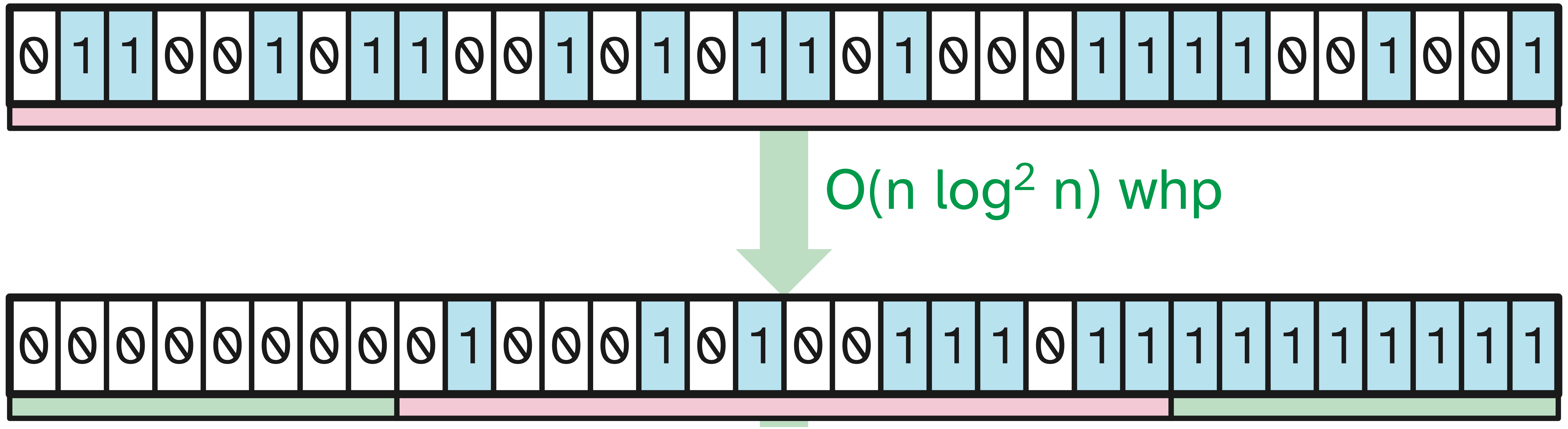


- Each iteration moves that bad 1 out of A[1...n/4] with probabilty 1/O(n log n).
- So on average we need O(n log n) iterations to move that bad 1 out of A[1..n/4]

Harmonic intuition

[Olesker-Taylor 2025]

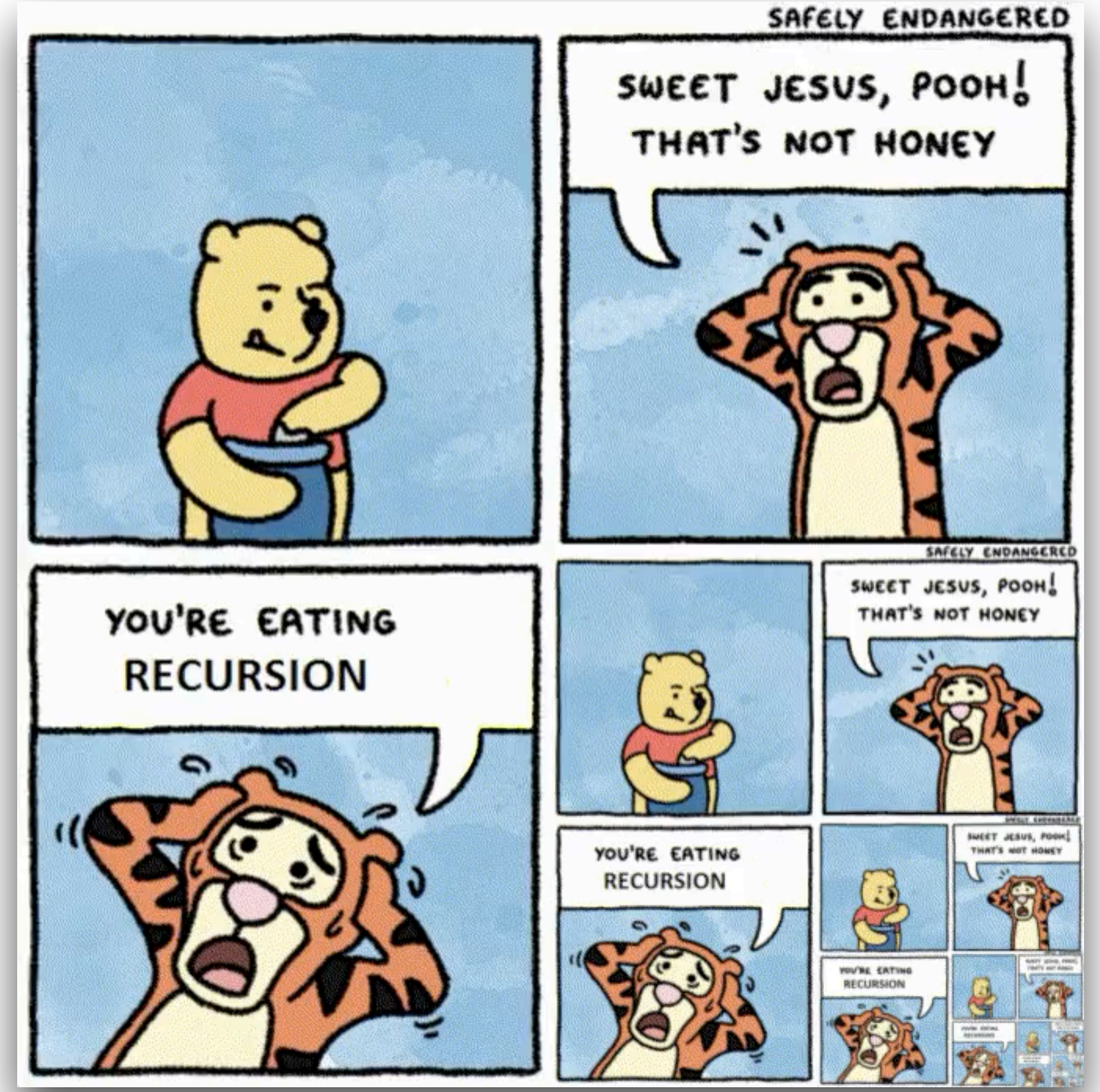
- With high probability, after O(n log² n) iterations, A[1…n/4] contains only 0s, and symmetrically, A[3n/4+1..n] contains only 1s.



Harmonic intuition

[Olesker-Taylor 2025]

- Now we only need to *recursively* sort the middle half A[n/4+1 .. 3n/4]!



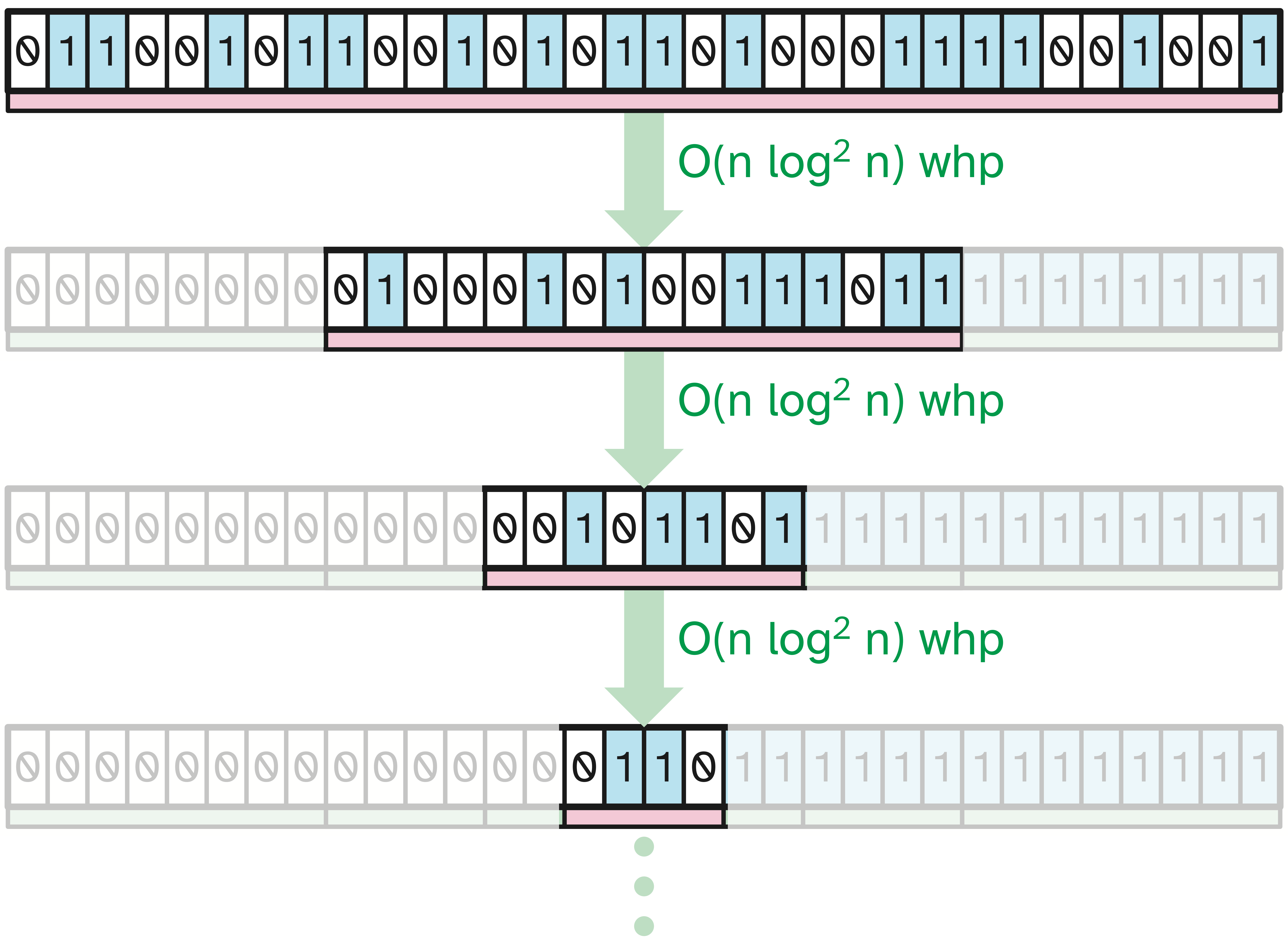
Harmonic intuition

[Olesker-Taylor 2025]

- The probabilities are *scale-invariant*. Moving a bad 1 out of the bottom quarter of A[n/4+1 .. 3n/4] takes another $O(n \log n)$ iterations.
- So every $O(n \log^2 n)$ iterations halve the unsorted part of the array.
- Recursion depth = $O(\log n)$, so the overall sorting time is $O(n \log^{\mathbf{3}} n)$.

Harmonic intuition

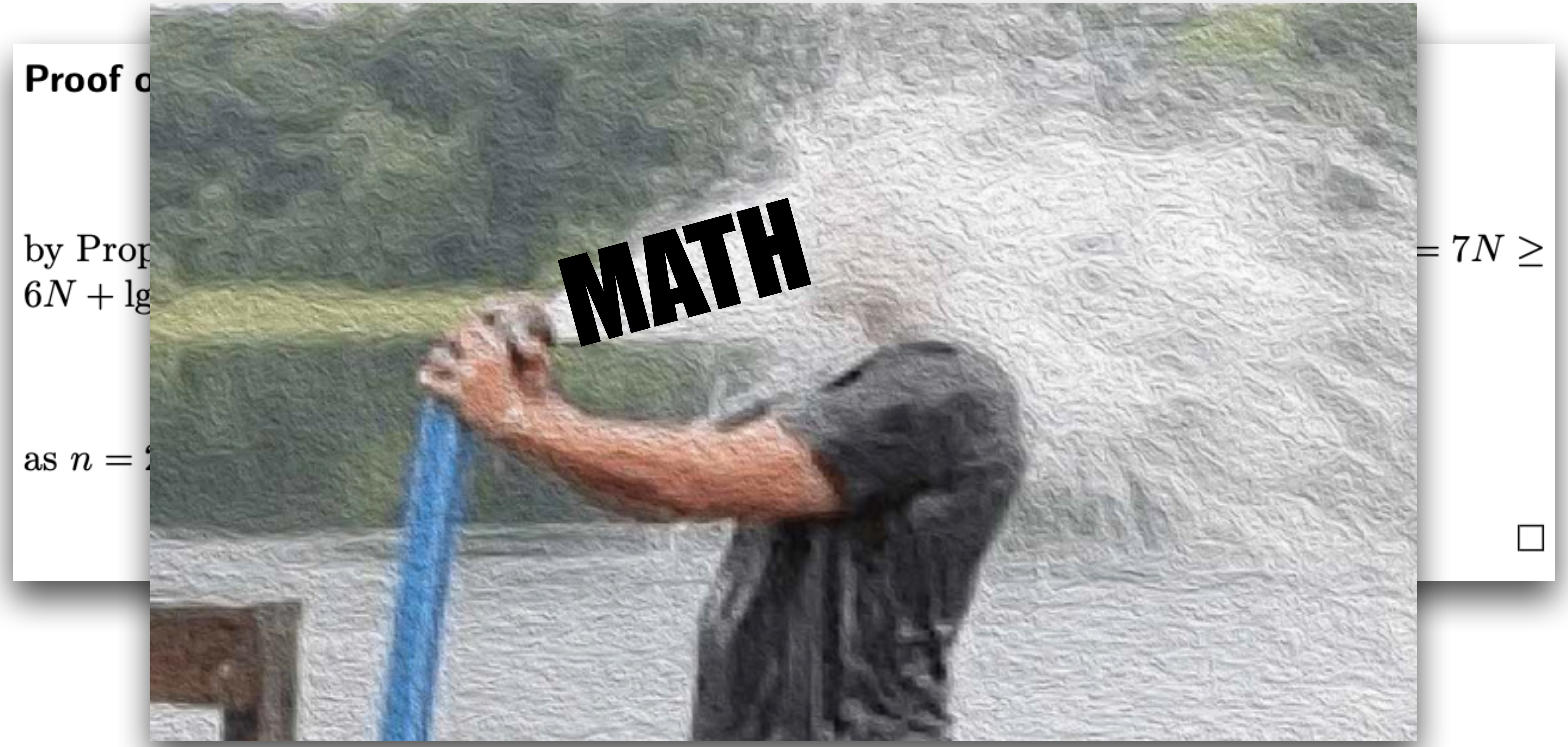
[Olesker-Taylor 2025]



Harmonic exchange

[Olesker-Taylor 2025]

The formal details, including removing the extra log factor, are unfortuntely complicated.



Open questions

[Olesker-Taylor 2025]

- Is there a simpler analysis? Please?
- Do other (simpler) distributions yield $O(n \log^2 n)$ time?
- Does *any* distribution yield $O(n \log n)$ time?
  - [Goodrich 2014] describes a data-oblivious algorithm that sorts using only $O(n \log n)$ *comparisons*, but it uses *expanders*, so it does not run in $O(n \log n)$ worst-case time.

Thank you!