# Heapsort

Using a heap to sort an array...

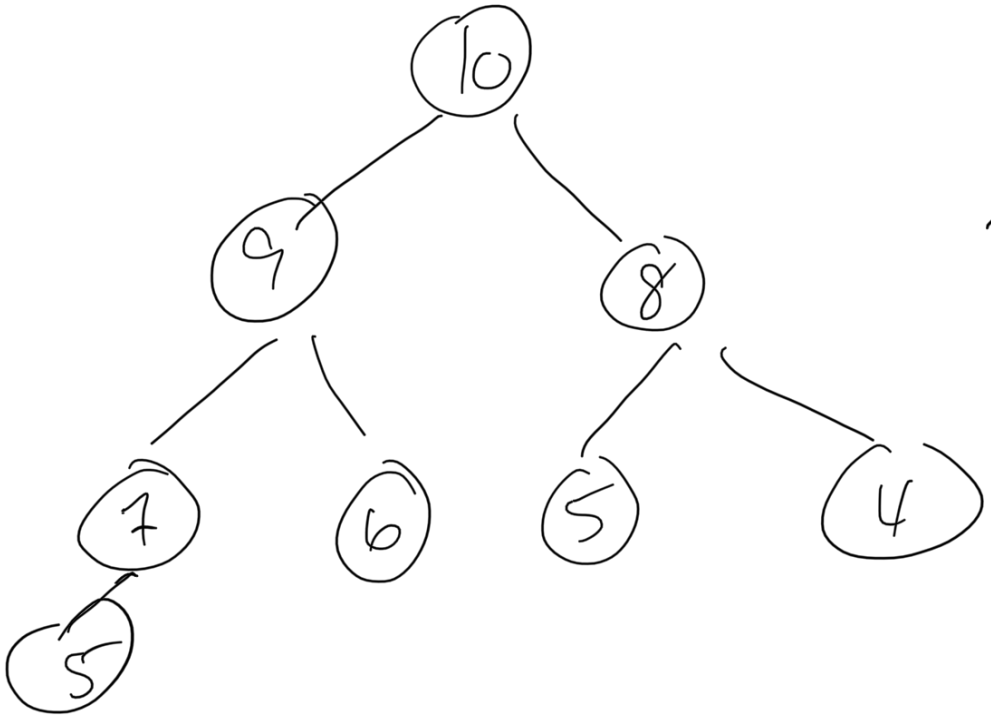
# Heapsort

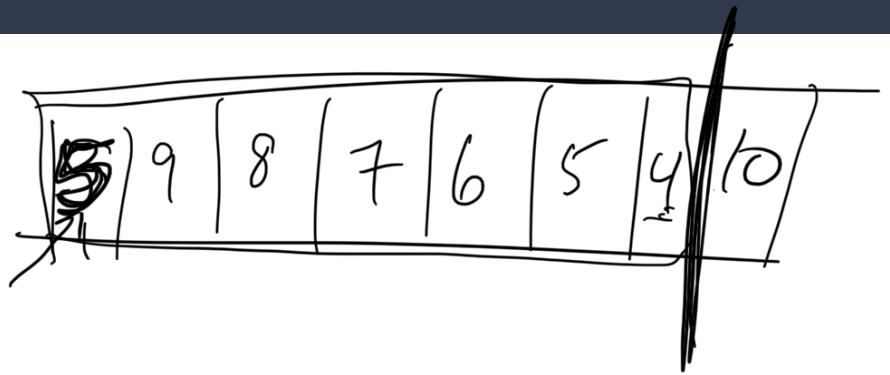
1. Heap Construction: Re-order array into heap

2. Sortdown: Pull items out of heap in decreasing order to build sorted array

**What kind of heap do we need–min or max?**
**How do we “construct” the heap?**
**How much time will this take?**
**Can we do Heapsort in place?**

# Heapsort: Overall Approach

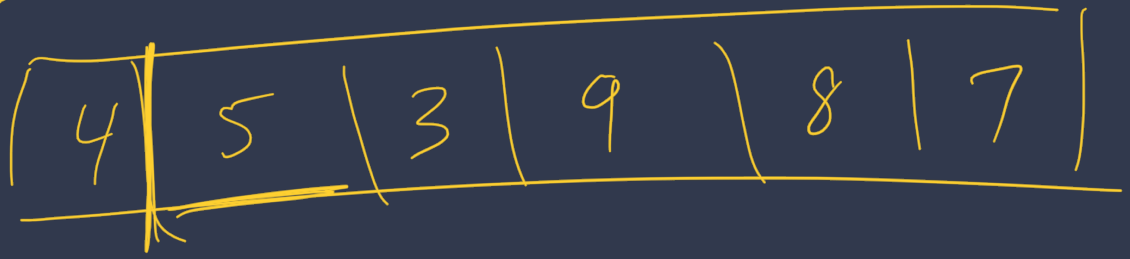


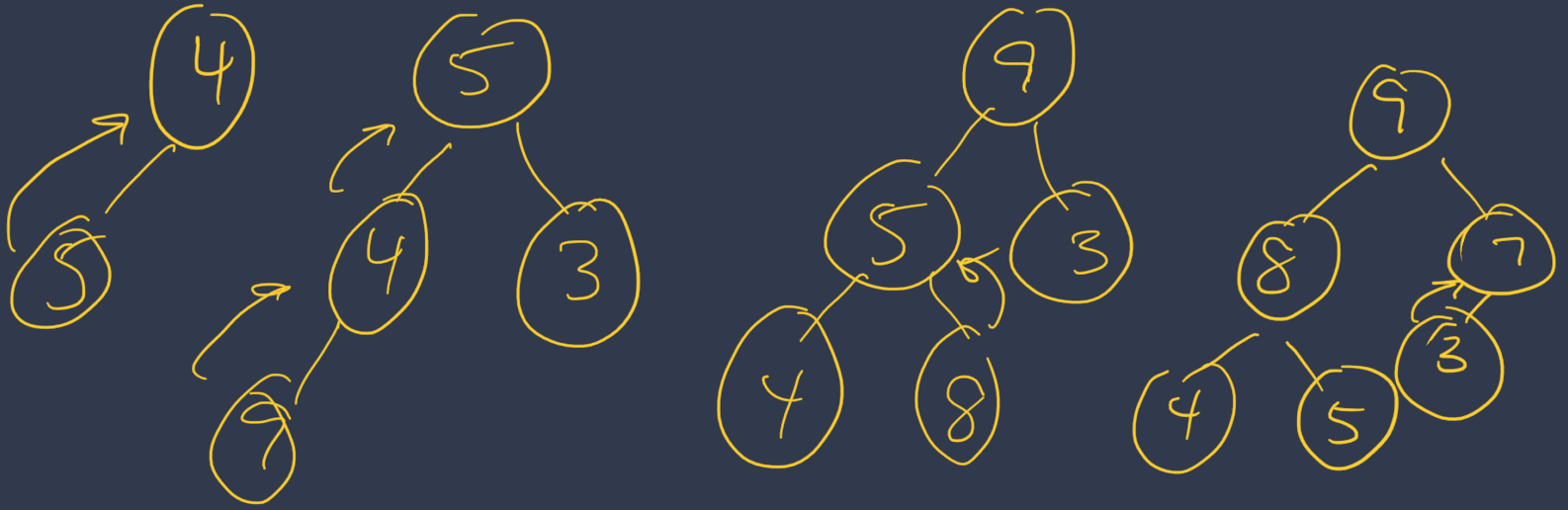



# Heap Construction

- There are two ways to put an array into max heap order.
  - Top-down, using swim: In this approach, you would iterate through each item in the array from left to right and “insert” each item into the growing heap.
  - Bottom-up, using sink: In this approach, you would iterate through the items from right to left and call sink on them–essentially you are combining smaller heaps together into bigger heaps until you just have one heap.

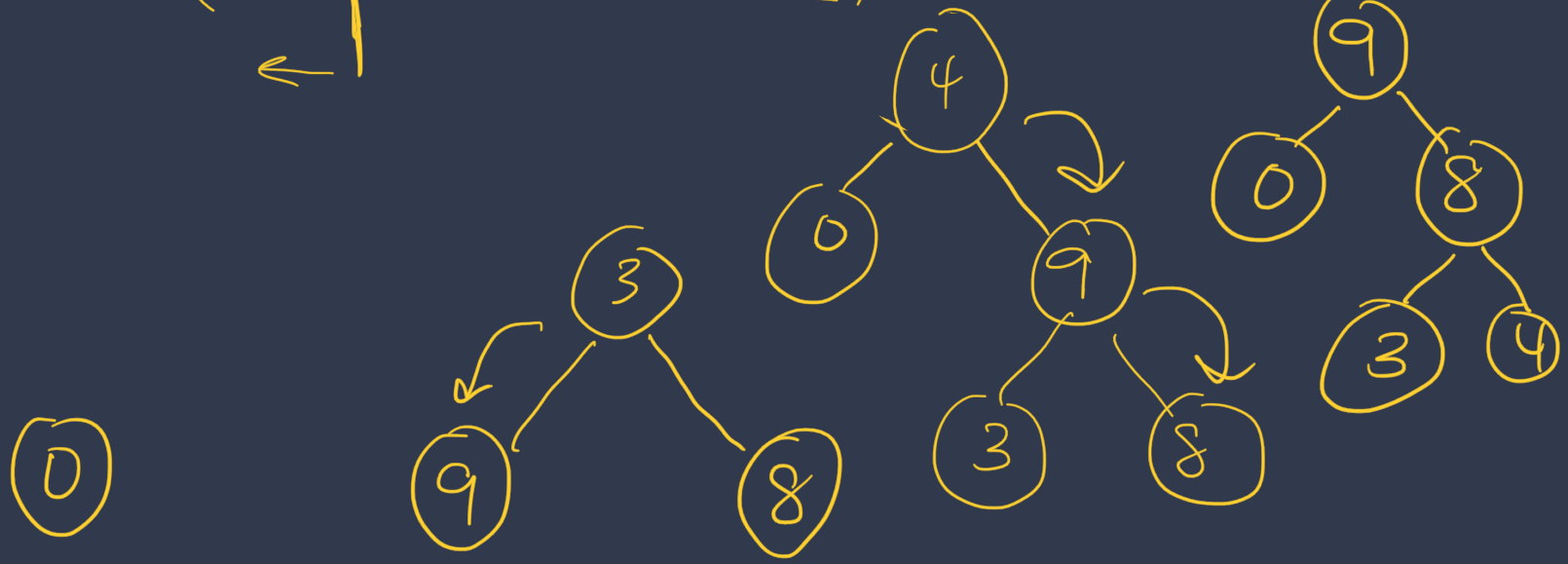
Top - Down




$\log 1 + \log 2 + \ldots + \log N \Rightarrow O(N \log N)$

# Bottom-up

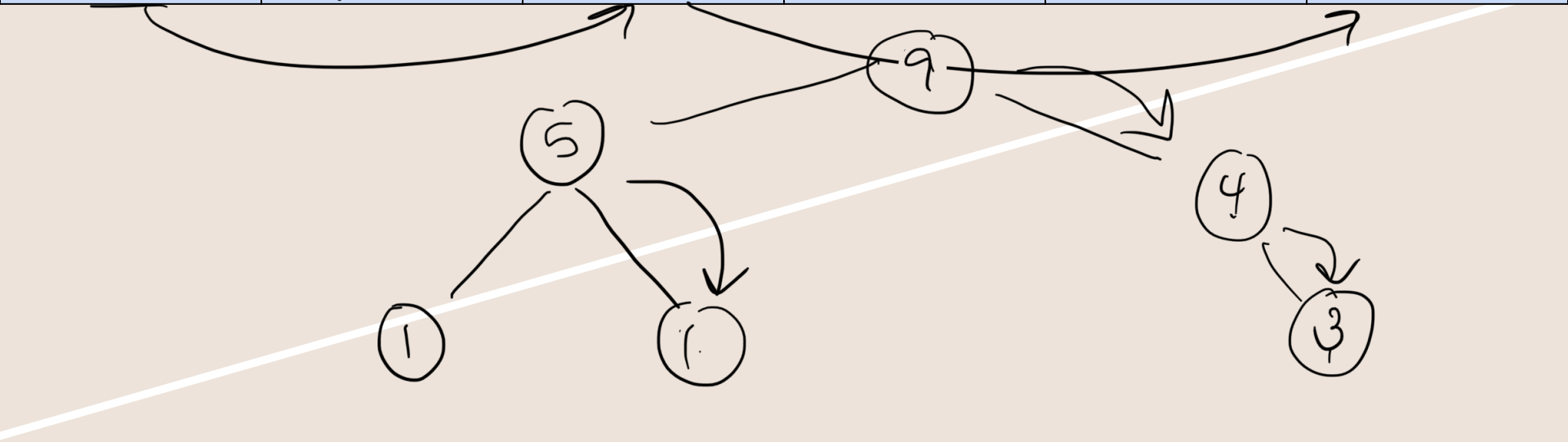



# Why bottom-up is better than top-down...

- It's faster: O(n) instead of O(nlogn).
- You have to call *sink* on about half the elements instead of calling *swim* on all the elements.
- The second part of Heapsort only requires the *sink* method, so doing the heap construction with *sink* means we don't have to implement *swim* at all (if we are only using the heap for sorting).

# Bottom up heap construction.

| 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 9 | 5 | 4 | 1 | 1 | 3 |



**Proposition R.** Sink-based heap construction uses fewer than $2N$ compares and fewer than $N$ exchanges to construct a heap from $N$ items.

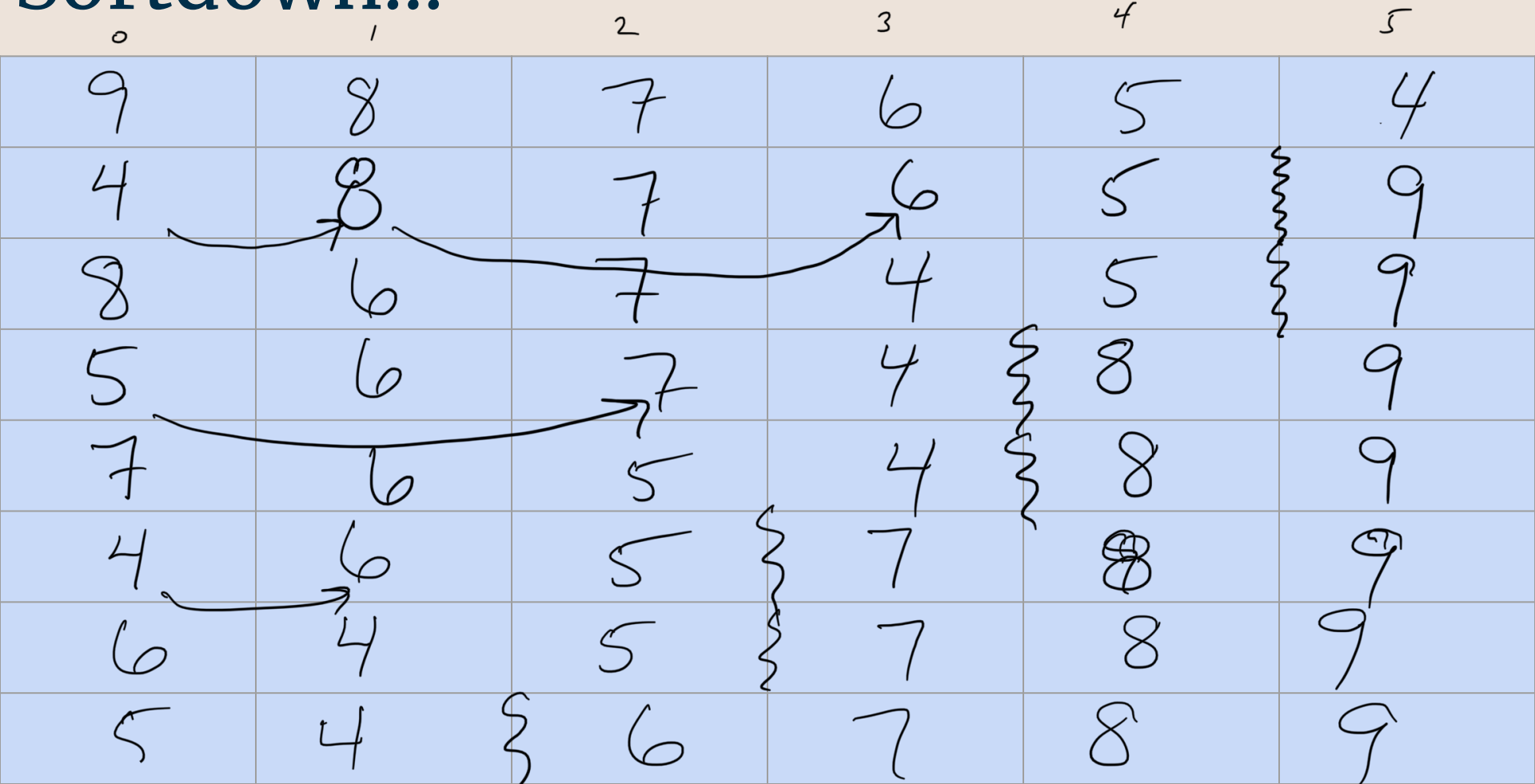
**Proof:** This fact follows from the observation that most of the heaps processed are small. For example, to build a heap of 127 items, we process 32 heaps of size 3, 16 heaps of size 7, 8 heaps of size 15, 4 heaps of size 31, 2 heaps of size 63, and 1 heap of size 127, so $32 \cdot 1 + 16 \cdot 2 + 8 \cdot 3 + 4 \cdot 4 + 2 \cdot 5 + 1 \cdot 6 = 120$ exchanges (twice as many compares) are required (at worst). See EXERCISE 2.4.20 for a complete proof.

Picture from [1]

# Sortdown: in-place

- Exchange the maximum with the last element (so the maximum is now in the right place for sorting the array, so we ignore it from now on, shrinking the size of the heap by 1).
- Then we reorder the heap and do it again.

# Sortdown...



# Runtime for Sortdown

- Exchanging the maximum is $O(1)$.
- Reordering the heap by calling *sink* on the first element is $O(\log m)$. ($m$ is size of heap)
- We exchange the maximum and reorder the heap $N$ times.

More specifically, the runtime is

$$1 + \log(N-1) + 1 + \log(N-2) + 1 + \log(N-3) + \dots + 1 + 1$$

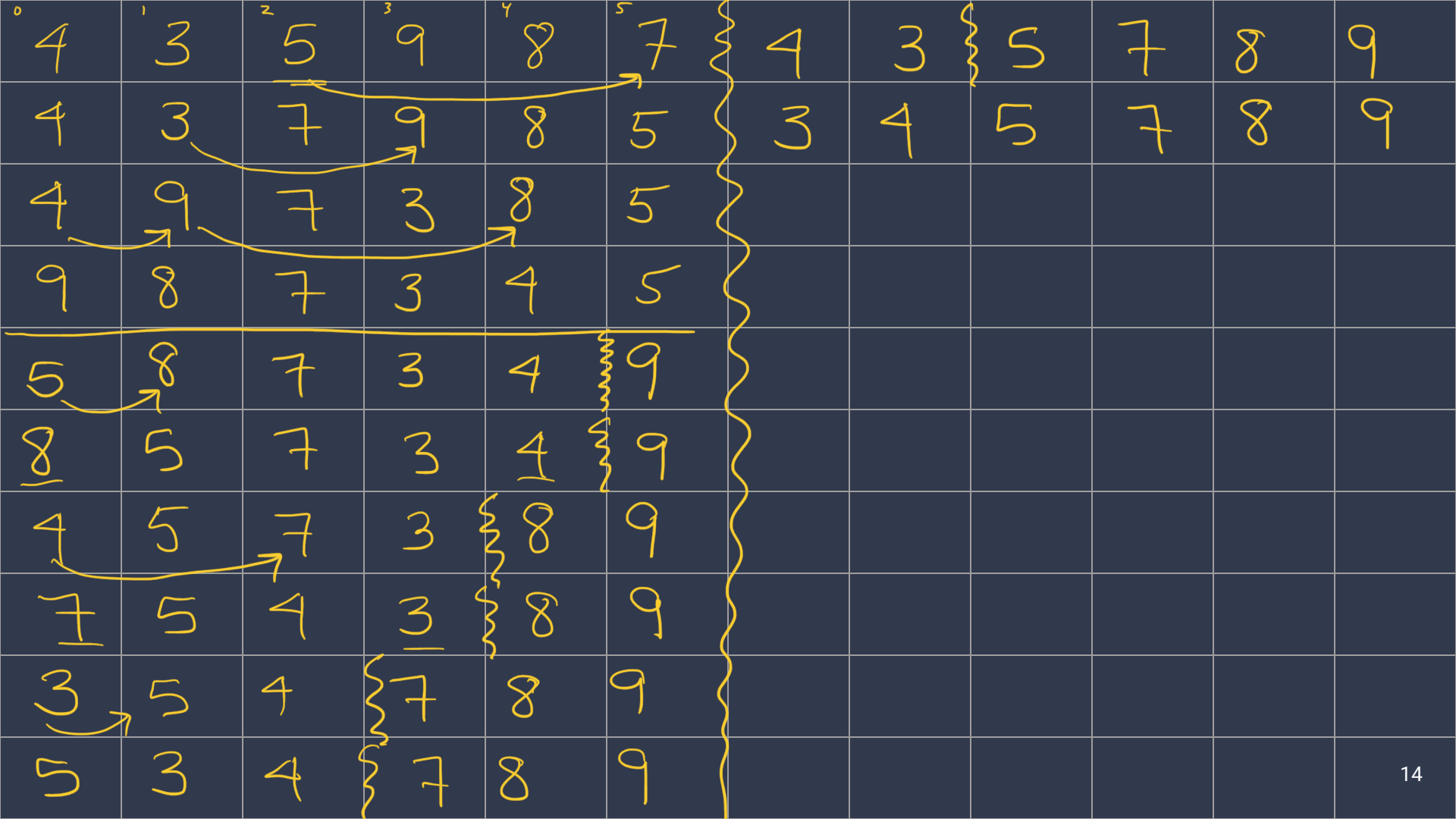
$$= \sum_{k=1}^{N-1} \log k + \sum_{k=1}^{N-1} 1 = O(N \log N) + O(N)$$

$\log(N-1)!$

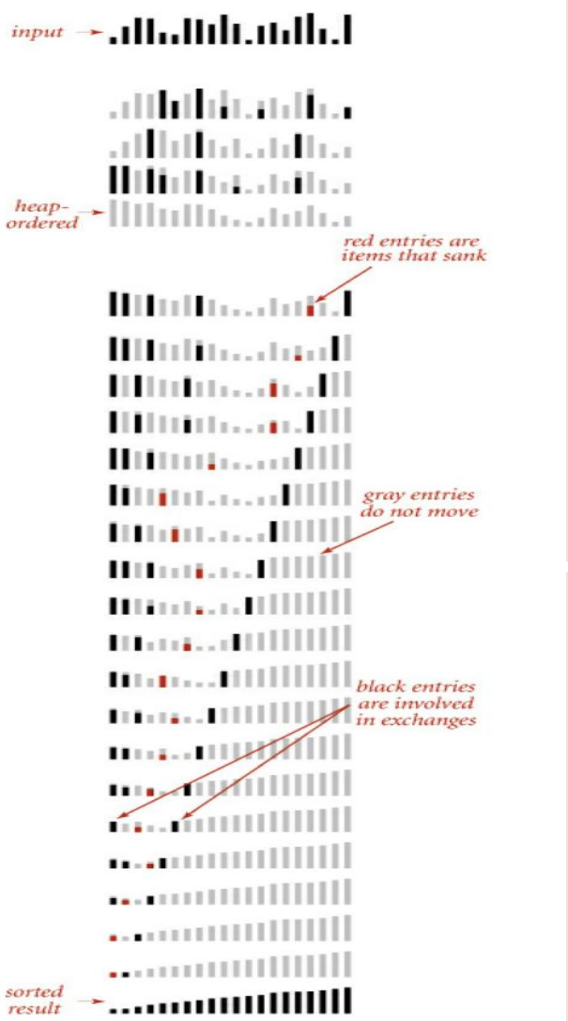
# Runtime for Heapsort

- Bottom-up Heap Construction: $O(N)$
- Sortdown: $O(N \log N)$

Total: $O(N \log N)$


0 1 2 3 4 5
4 3 5 9 8 7
4 3 7 9 8 5
4 9 7 3 8 5
9 8 7 3 4 5
5 8 7 3 4 9
8 5 7 3 4 9
4 5 7 3 8 9
7 5 4 3 8 9
3 5 4 7 8 9
5 3 4 7 8 9
4 3 5 7 8 9
3 4 5 7 8 9

# Heapsort Pseudocode




Visual trace of heapsort

Picture from [1]

# Heapsort Summary

- Runtime: O(NlogN)
- Popular when space is tight (e.g. embedded systems) because it provides optimal performance with just a few dozen lines of code
- Not often used in typical applications in modern systems because of its poor cache performance (array entries are rarely compared with nearby entries causing more cache misses)
- To do this in-place:
  - to sort from smallest to largest, use a max-heap
  - to sort from largest to smallest, use a min-heap

# References

[1] *Algorithms, Fourth Edition;* Robert Sedgewick and Kevin Wayne (and associated slides)
[2] Book slides from Goodrich and Tamassia