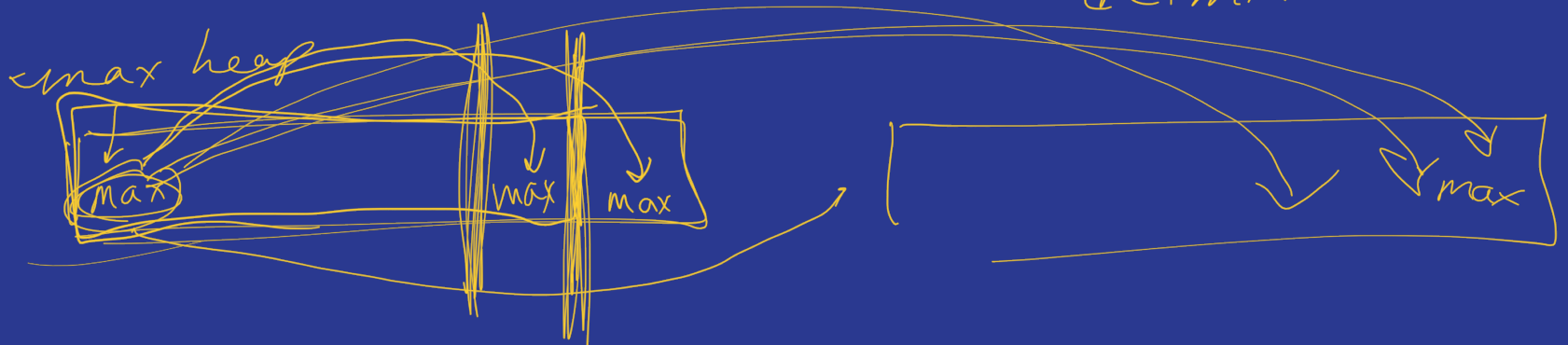
Sorting III

Using a heap to sort an array...

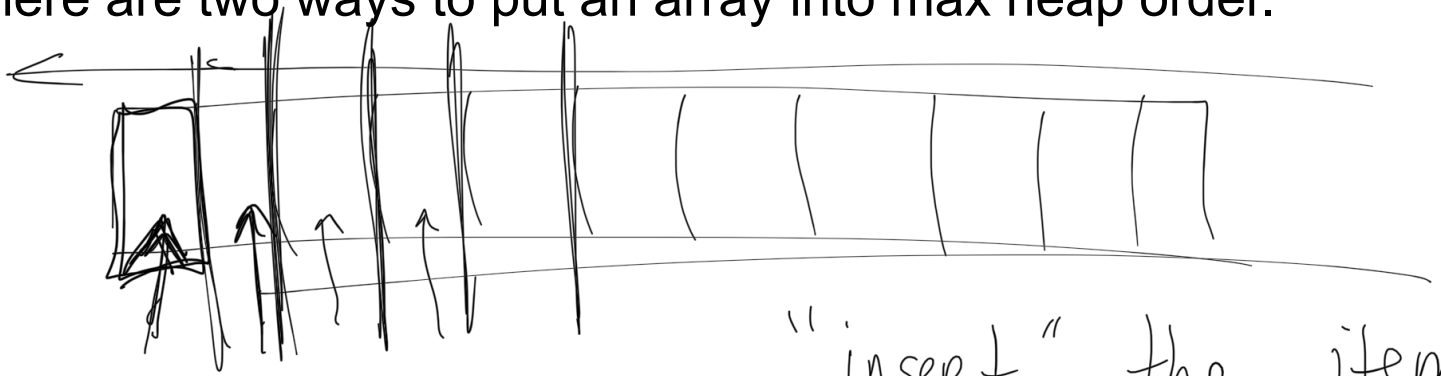


Heapsort: Overall Approach

Heap Construction

top-down

There are two ways to put an array into max heap order.



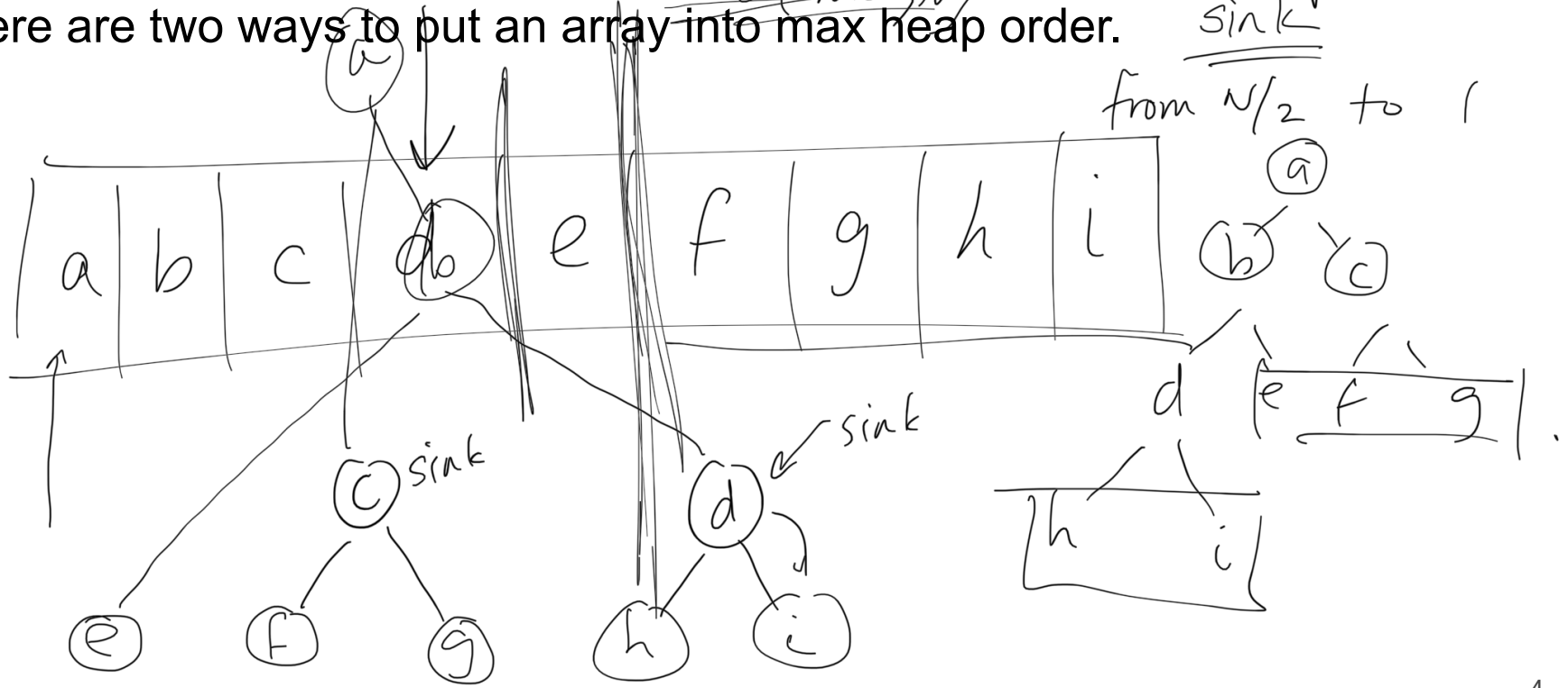
"insert" the items from left to right → swim

swim: $O(\log N)$

total time: $O(N \log N)$

Heap Construction

There are two ways to put an array into max heap order.

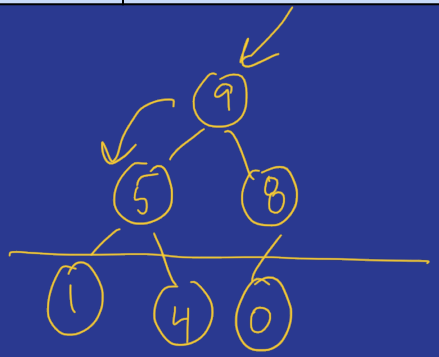


Why bottom-up is better than top-down…

- It's faster: O(N) instead O(N log N)
- 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	8	1	4	0



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

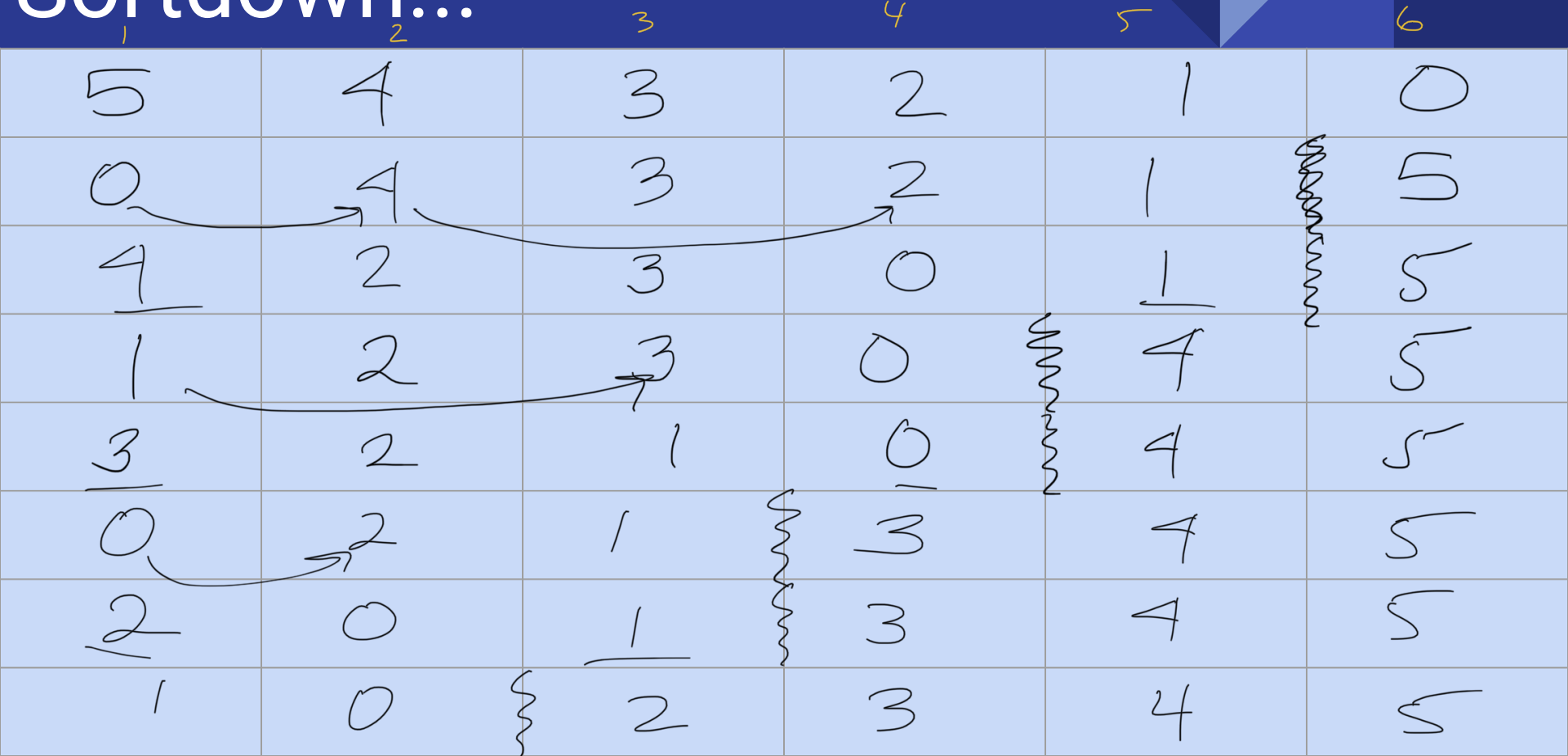
Proof: This fact follows from the observation that most of the heaps processed are small. For example, to build a heap of 127 items, we process 32 heaps of size 3, 16 heaps of size 7, 8 heaps of size 15, 4 heaps of size 31, 2 heaps of size 63, and 1 heap of size 127, so $32 \cdot 1 + 16 \cdot 2 + 8 \cdot 3 + 4 \cdot 4 + 2 \cdot 5 + 1 \cdot 6 = 120$ 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 _max_ 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 using _sink_ on the first element and do it all again.

Sortdown...



Runtime for Sortdown

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

$O(N \log N)$

More specifically, the runtime is

$$\log(N-1) + \log(N-2) + \ldots + \log 1$$

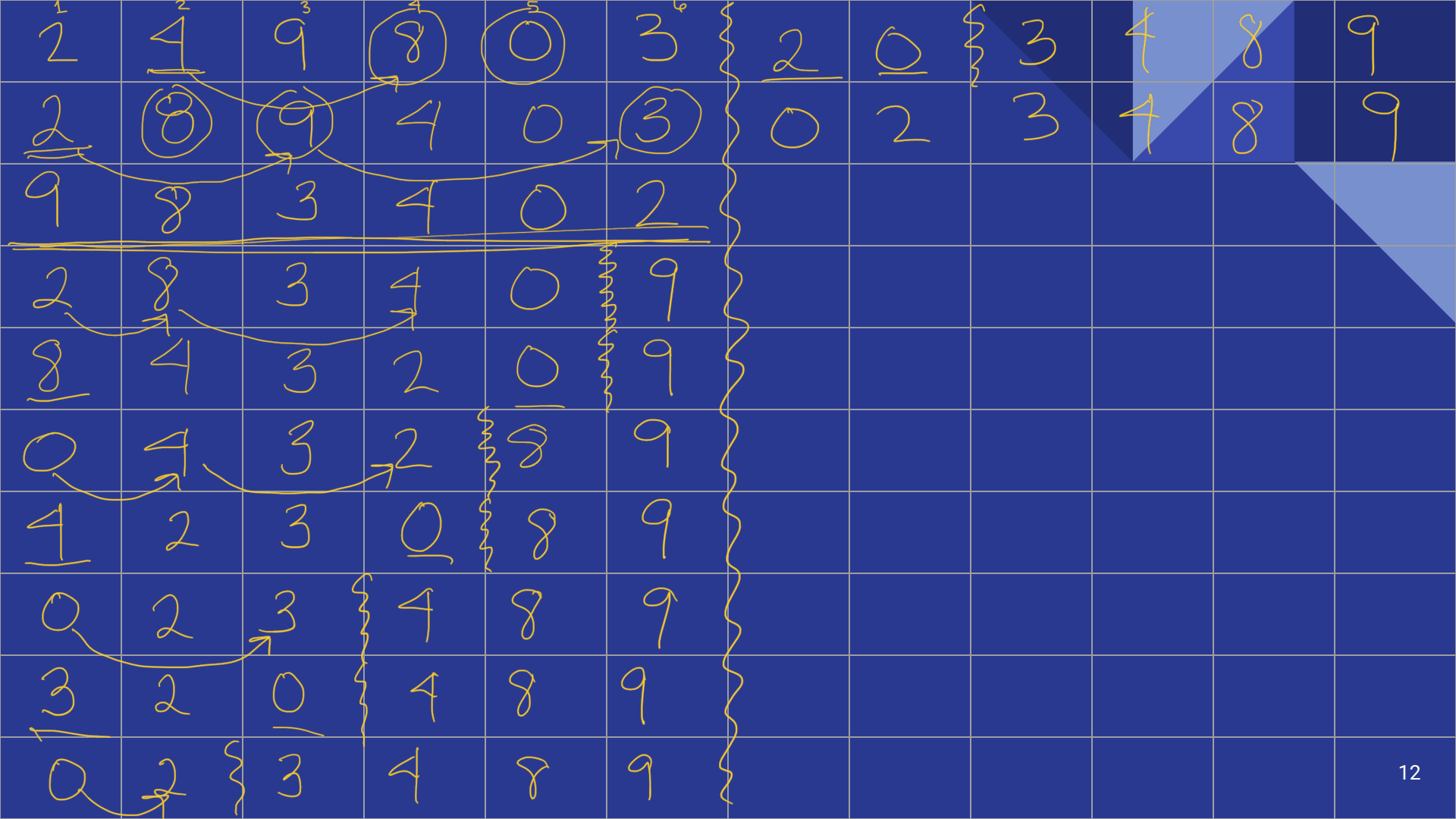
$$= \log(N-1)! \longrightarrow O(N \log N)$$

Runtime for Heapsort

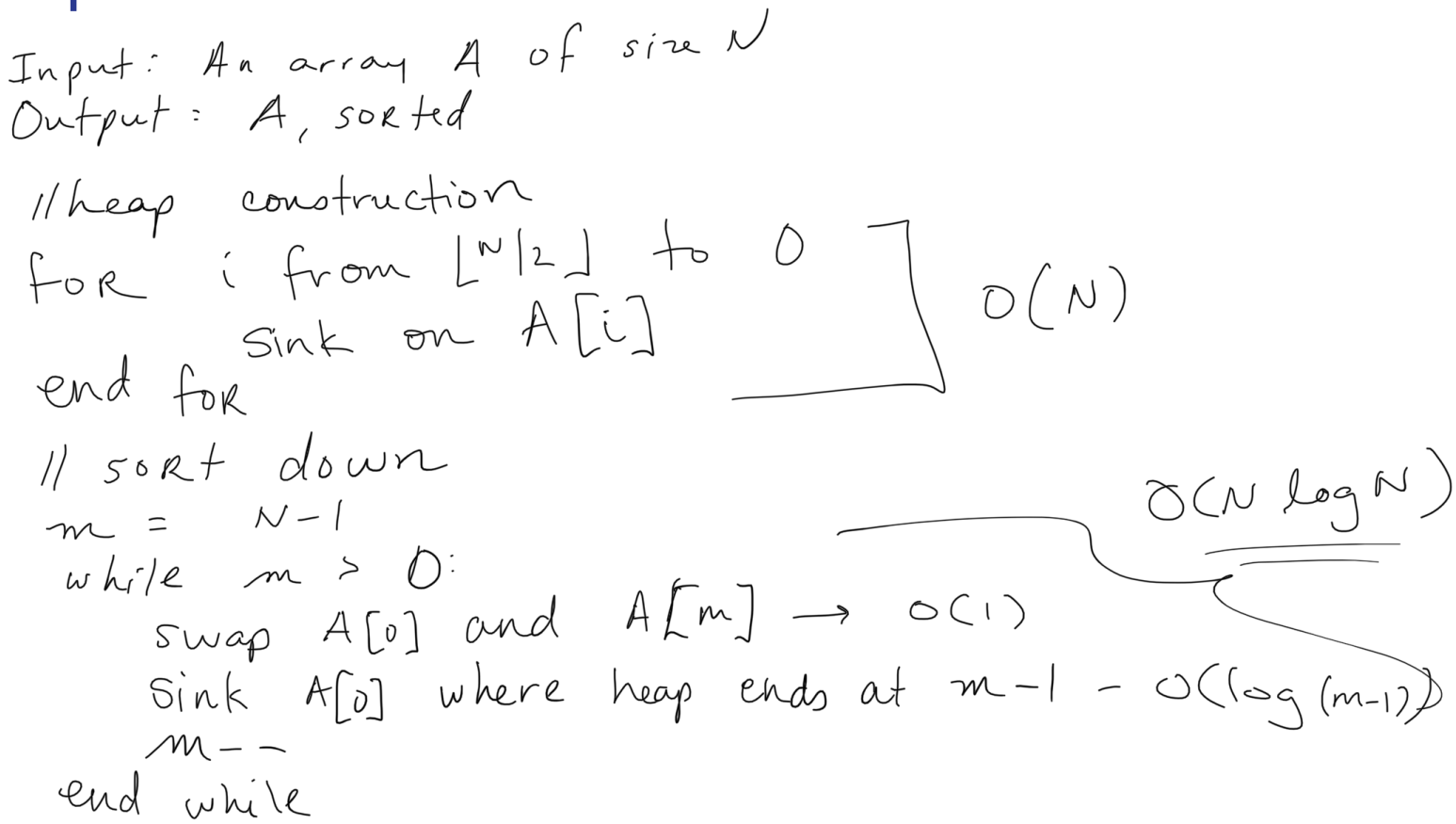
- Bottom-up Heap Construction: $O(N)$
- Sortdown: $O(N \log N)$

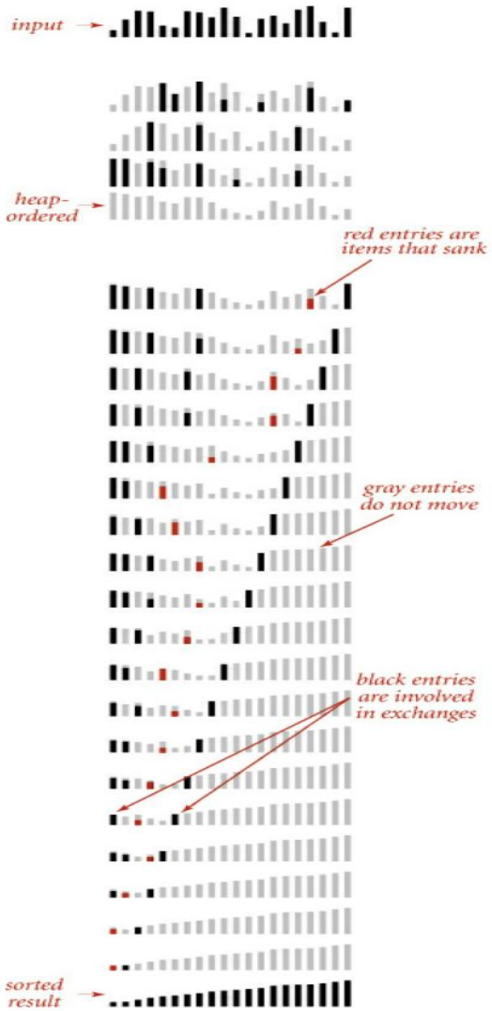
Total: $O(N \log N)$ → both best and worst case

trivial best case - everything is the same: $O(N)$


2 4 9 8 0 3
2 8 9 4 0 3
9 8 3 4 0 2
2 8 3 4 0 9
8 4 3 2 0 9
0 4 3 2 8 9
4 2 3 0 8 9
0 2 3 4 8 9
3 2 0 4 8 9
0 2 3 4 8 9
2 0 3 4 8 9
0 2 3 4 8 9

Heapsort Pseudocode





Visual trace of heapsort

Picture from [1]

Heapsort Summary

- Runtime: $O(N \log N)$
- 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)