Comparison Sorts:

For each:

1. How it is implemented? — Stable? / space complexity?
2. Best case and worst case runtimes, and **why**!
3. Why would we use it?

Selection Sort:

→ move minimum of unsorted to the left

7	1	9	4	3	5	6	2

1	7	9	4	3	5	6	2

1	2	9	4	3	5	6	7

1	2	3	4	9	5	6	7

Runtime:

Best Case: $\Theta(N^2)$ $\Rightarrow N + N-1 + N-2 + \ldots + 1$

(finding first minimum: N; finding second minimum: $N-1$)

Worst Case: $\Theta(N^2)$

Why use?

1. Simple to code
2. Minimize swapping operations, only N swaps
3. Pretty bad overall though

Understanding

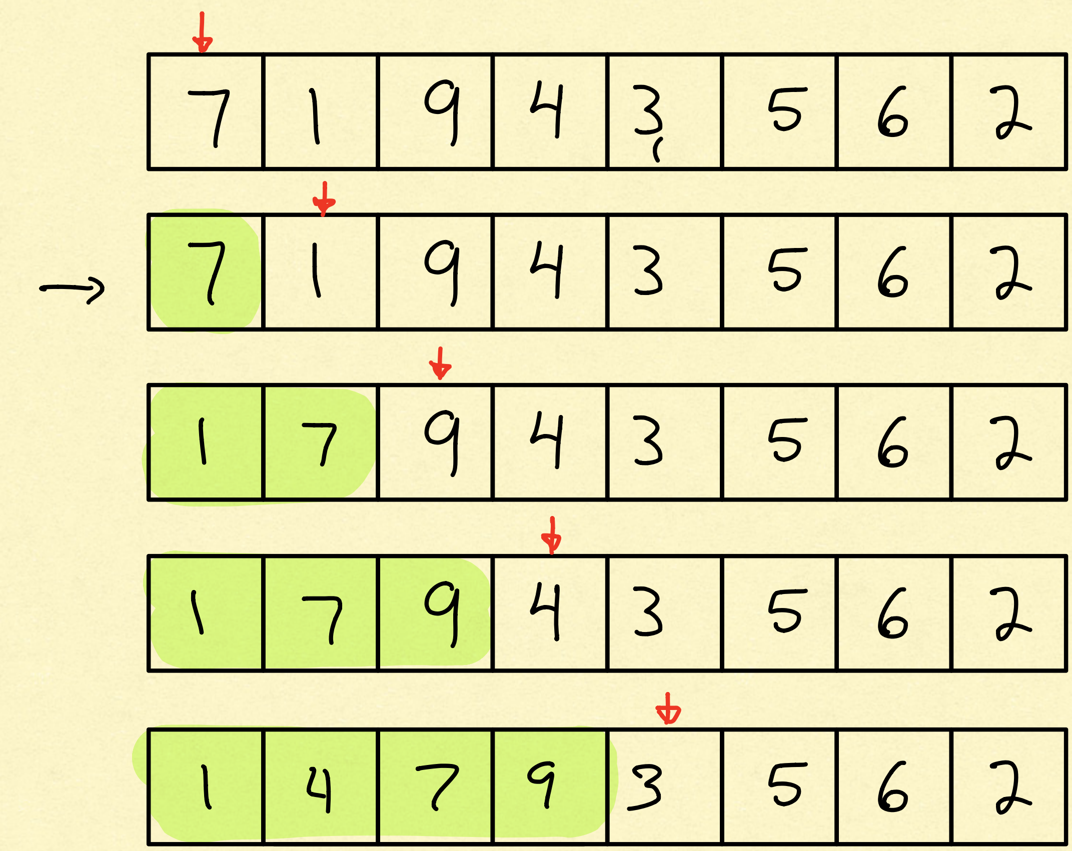
Question: Is selection sort stable?

Stability: Two elements that are the equal are ordered the same in final result as in original list

ANSWER: Yes, if selection sort is implemented such that we only change the current minimum if we find something strictly less

INSERTION SORT:

→ Build sorted sublist on left side of list



RUNTIME:

Best Case: $\Theta(N)$ → list sorted in ascending order

Worst Case: $\Theta(N^2)$ → list sorted in descending order

WHY USE?

1. Very fast on small inputs ($n < 15$)
2. Very fast when the input list is nearly sorted

UNDERSTANDING

QUESTION: The runtime of insertion sort can be written as $\Theta(N + k)$

k = # of <u>inversions</u>. Why?

Recall that an inversion is a pair of elements (x, y) where x precedes y but is <u>greater</u> than y.

E.g. $(7, 3)$ is an inversion because $7 > 3$

1	7	3	2

ANSWER: We can think of K as the total number of swapping operations that need to be done, i.e. for every inversion, we need one swap!

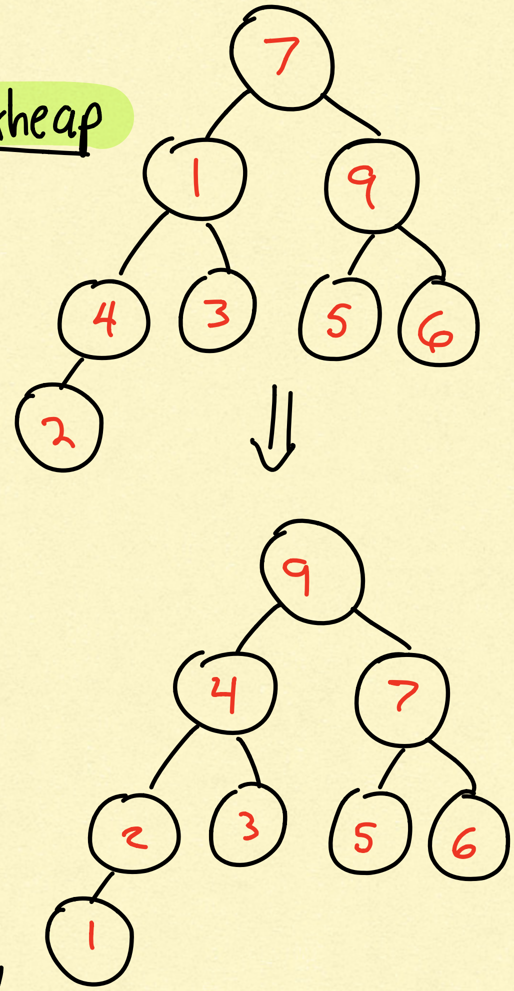
1	7	3	2

Looking at the array above, notice that 2 is the right element in two pairs – $(7,2)$, $(3,2)$. Accordingly, when we move 2 to the left, we need 2 swaps!

HEAPSORT:

1) Bottom up heapification to build a maxheap

→ bubble down everyone from the end



7	1	9	4	3	5	6	2

heapify

9	4	7	2	3	5	6	1

2) Repeat **N** times:

→ remove**Max** and put **max** at end

→ bubble down to create smaller maxheap while growing sorted array from righthand side of array

9	4	7	2	3	5	6	1

removeMin() → swap 9 and 1

1	4	7	2	3	5	6	9

bubble down 1

7	4	6	2	3	5	1	9

removeMin() → swap 9 and 1

1	4	6	2	3	5	7	9

bubble down 1

6	4	5	2	3	1	7	9

Runtime:

Best Case: $\Theta(N)$ → all duplicates

Worst Case: $\Theta(N \log N)$ → Heapify: $\Theta(N)$; Bubble down operations: $\Theta(N \log N)$

Why use?

1. Good worst case bound
2. If we already are given a heap
3. In place, i.e. constant space complexity!

Understanding

Question: Suppose you had a magical maxheap with constant time for bubbleDown, how would the worst case runtime change, if at all?

Answer: $\Theta(N)$! We have N bubble down operations, each taking $\Theta(1)$ time $\Rightarrow \Theta(N)$.

MergeSort

→ keep merging runs together, starting with runs of size zero/one.

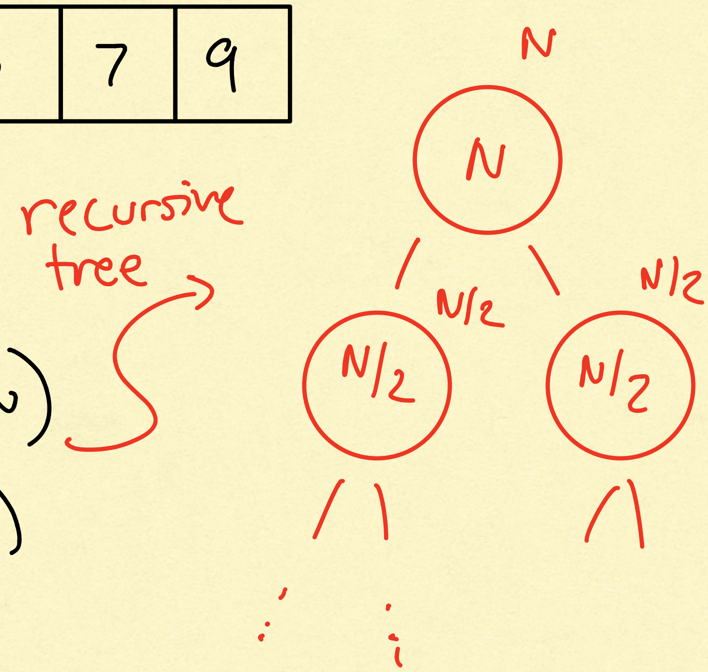
7	1	9	4	3	5	6	2

7	1	9	4	3	5	6	2

1	7	4	9	3	5	2	6

1	4	7	9	2	3	5	6

1	2	3	4	5	6	7	9



Runtime:

Best Case: $\Theta(N \log N)$

Worst Case: $\Theta(N \log N)$

Why use?

→ good worst case bound

→ stable ⇒ often used w objects

→ good with linked lists

Understanding Question: How could you change mergesort with as little work as possible to lose stability?

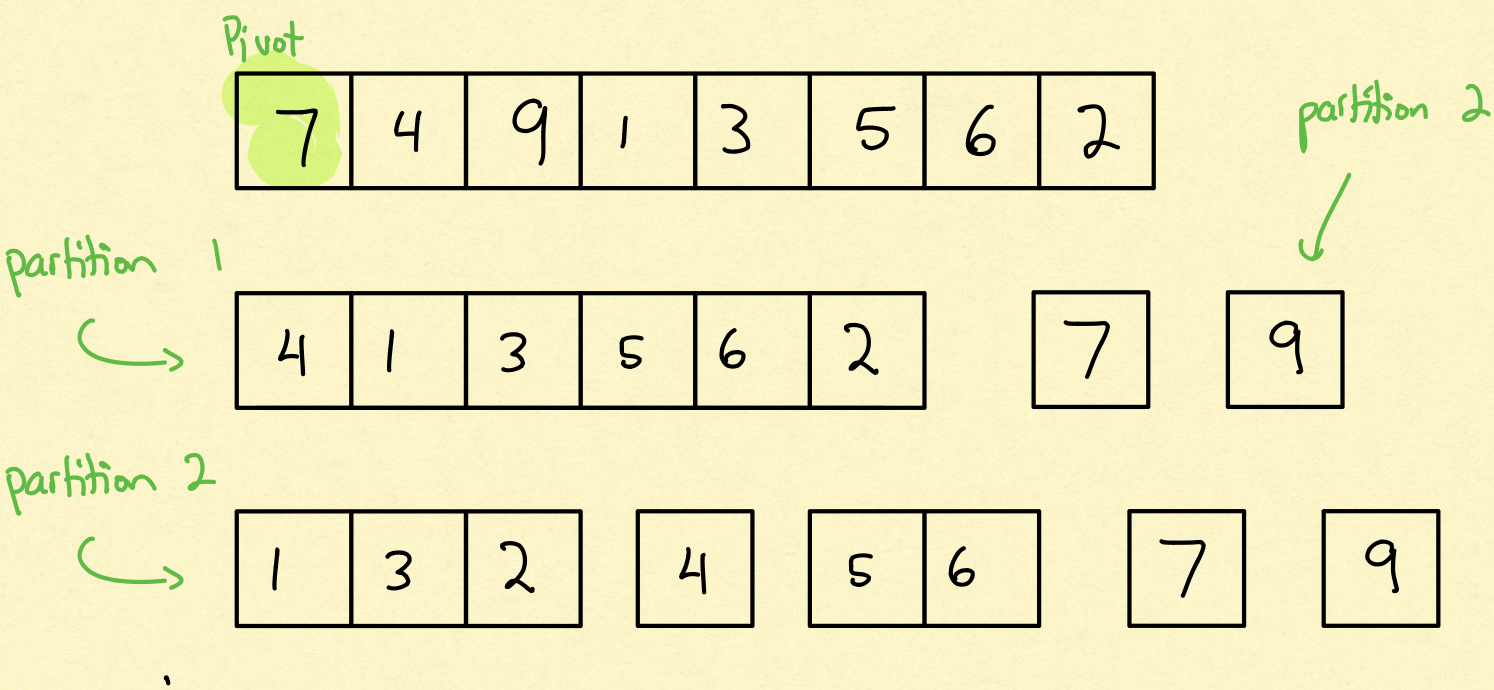
Answer: First, let's recall why merge sort is stable. In the merging process, we break ties by choosing the element from the left sublist! Now, to lose stability, we can simply break ties by choosing from the right!

Quicksort

3 Scan:

Choose a pivot and partition into 3 groups those smaller than pivot, equal, and greater

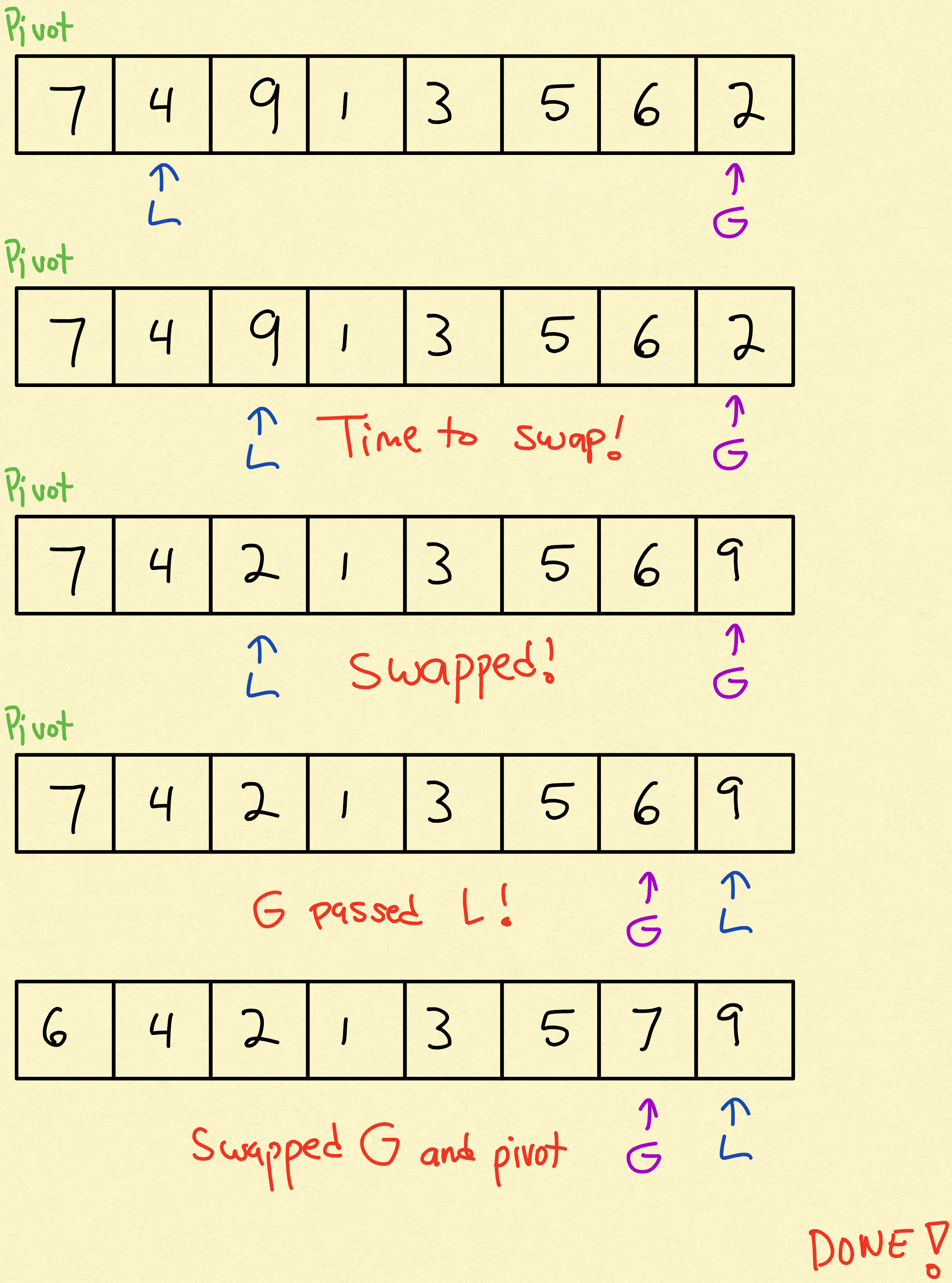
→ recurse on each group



⋮

Hoare Partitioning:

- 2 pointers L & G that start at left & right ends
  - L likes small items and G likes big ones
- Move pointers to eachother, stopping on disliked item.
  - If both stopped, swap, and move pointers
- Done when pointers cross, then swap G & pivot



Why use?

- → The fastest sort!
- → In place with Hoare
- → Stable with 3 scan

Runtime:

Best Case: $\Theta(N\log N)$ → if we choose a good pivot each time

Worst Case: $\Theta(N^2)$ → if the list is given in sorted order

Understanding

Question: Hoare Partitioning may not be stable, but can it be? Show an example of a 5 element list with duplicates where Hoare Partitioning is stable, or prove it cannot be done. Recall that both L and G stop on equal elements.

Answer:

3	1	1	1	2

As shown in the example above, just because a sorting algorithm is not stable, it doesn't mean it *always* breaks stability while sorting!