CSC148 – Introduction to Computer Science

Lecture 11: Introduction to Sorting (continued)

Sean Henderson

Quicksort

- Like merge sort, this algorithm will revolve around some operation we can perform in linear time, and recursively sorting two parts of the list

Partition

- Given a list A[0:n], and an value pivot, modify A such that for some i, everything in A[0:i] < pivot and everything in A[i:n] >= pivot
- Everything less than pivot is at the beginning of the list, everything else is at the end

Partition Example

- Say A = [1, 4, 2, 3, 0] and pivot = 3
- The we want everything less than 3 at the start of the list, then everything greater than or equal to 3
- For example, [1, 2, 0, 4, 3] satisfies this

How to Partition

- Easy; sort the list
  - It is then, by definition, partitioned
  - O(n log n) runtime, using merge/heap sort
  - Obviously we would like to do better
- Come up with a linear way of performing this operation

Partition

- Create two lists, X and Y
- Iterate over every element x in A
- If x < pivot, append it to X
- If x >= pivot, append it to Y
- Finally, copy X to the first len(X) positions in A, and Y to the last len(Y) positions

Partition

- Problem: that algorithm is not in-place (we would like it to be)
- Recall: in-place (basically) means we don't make copies of the list to work with, like merge sort does
- We would like to design an in-place partition algorithm

Partition

- Idea similar to heap sort
- The first chunk of our list will store X (elements less than pivot, initially empty)
- The rest of our list will store Y
- For convenience, our method will return the length of X (although this could be found after the fact with a linear scan)

Partition

- Example

1	2	0	4	3	2.5

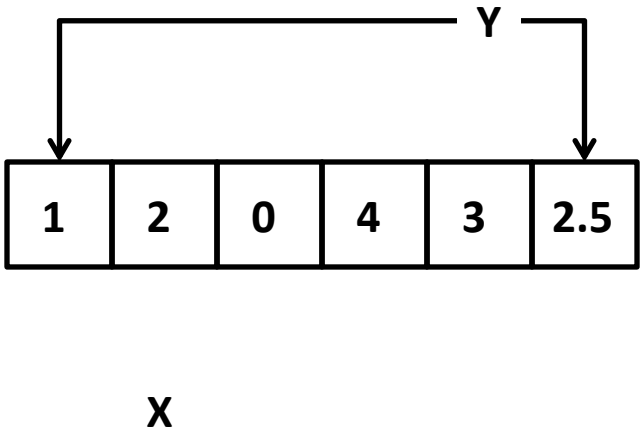
Partition

- Perform in-place partition
- Build X and Y
- Initially, everything is in Y (things >= pivot) and we iterate through to move things to X

1	2	0	4	3	2.5

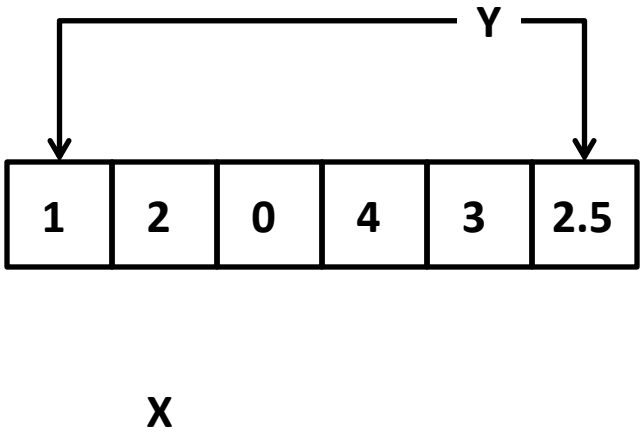
Partition

- X starts off empty
- Y is everything



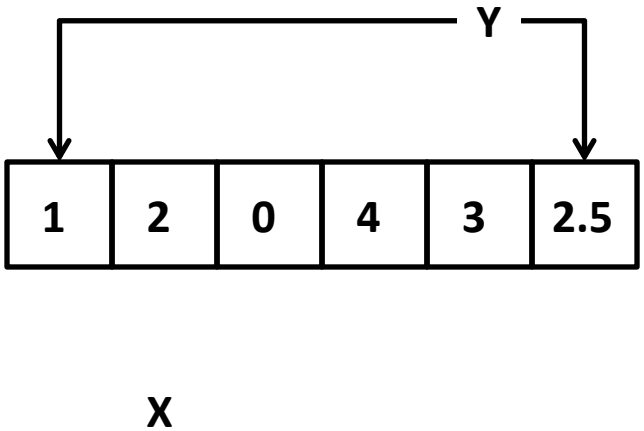
Partition

- X starts off empty
- Y is everything
- Take pivot = 3



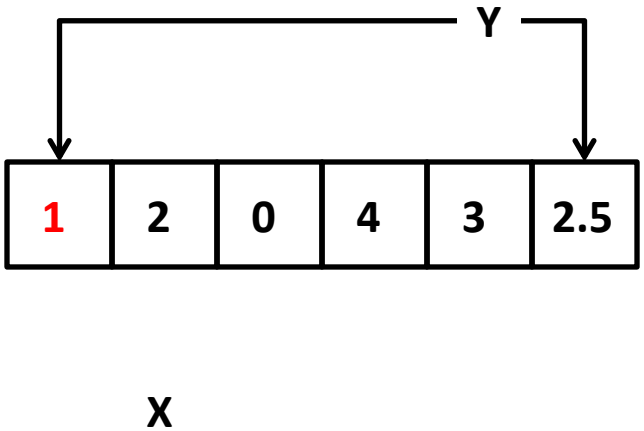
Partition

- X starts off empty
- Y is everything
- Take pivot = 3
- Iterate through our list, looking for things in Y that should be in X



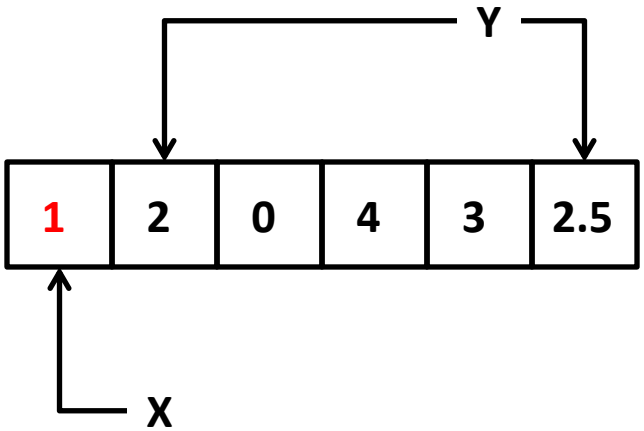
Partition

- X starts off empty
- Y is everything
- Take pivot = 3
- Iterate through our list, looking for things in Y that should be in X



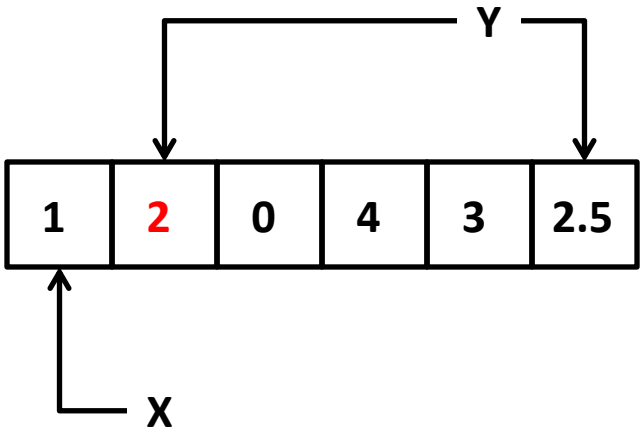
Partition

- X starts off empty
- Y is everything
- Take pivot = 3
- Iterate through our list, looking for things in Y that should be in X



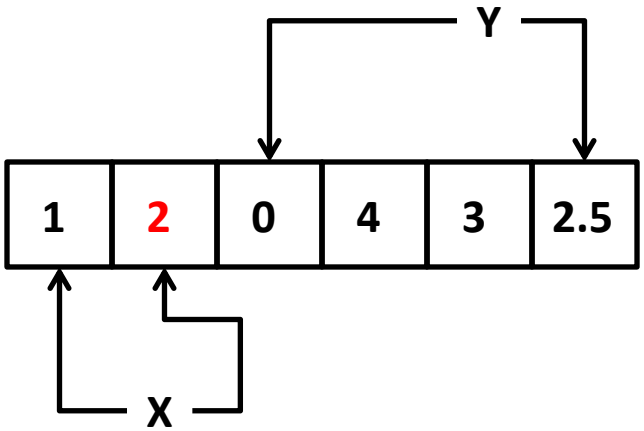
Partition

- X starts off empty
- Y is everything
- Take pivot = 3
- Iterate through our list, looking for things in Y that should be in X



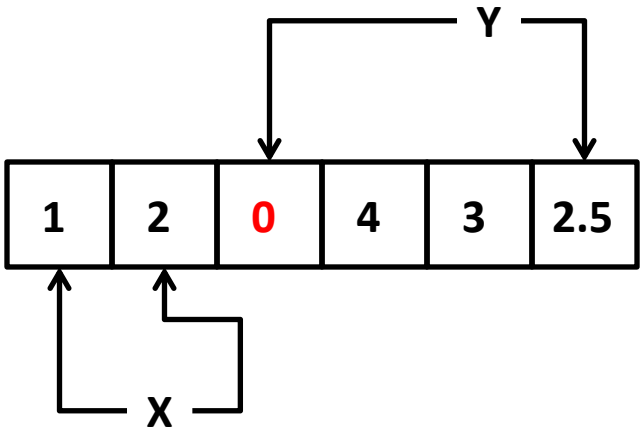
Partition

- X starts off empty
- Y is everything
- Take pivot = 3
- Iterate through our list, looking for things in Y that should be in X



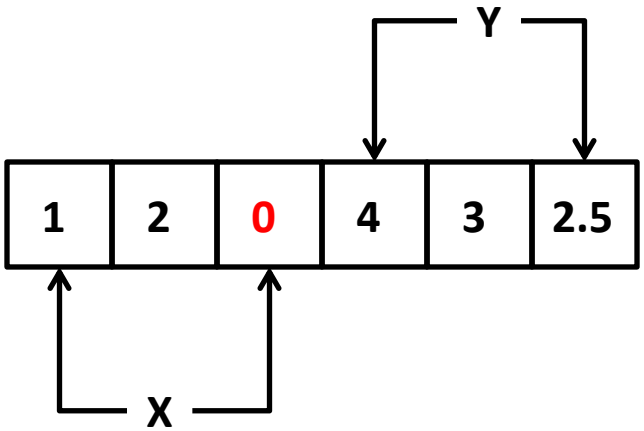
Partition

- X starts off empty
- Y is everything
- Take pivot = 3
- Iterate through our list, looking for things in Y that should be in X



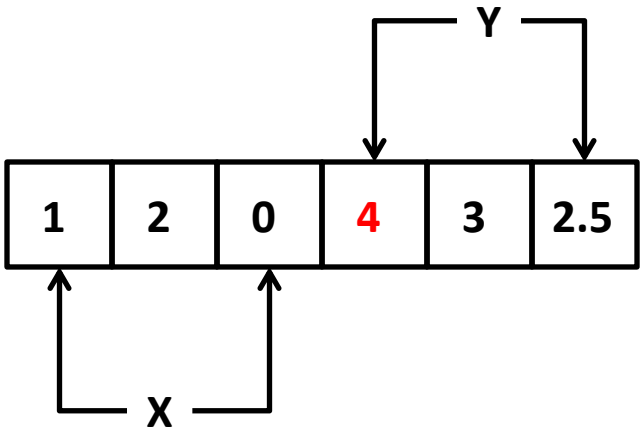
Partition

- X starts off empty
- Y is everything
- Take pivot = 3
- Iterate through our list, looking for things in Y that should be in X



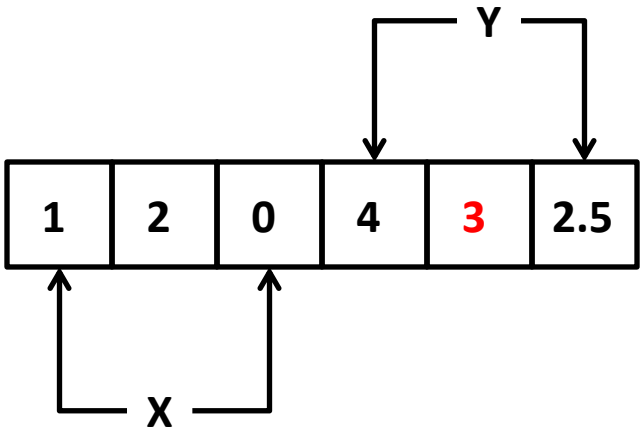
Partition

- X starts off empty
- Y is everything
- Take pivot = 3
- Iterate through our list, looking for things in Y that should be in X



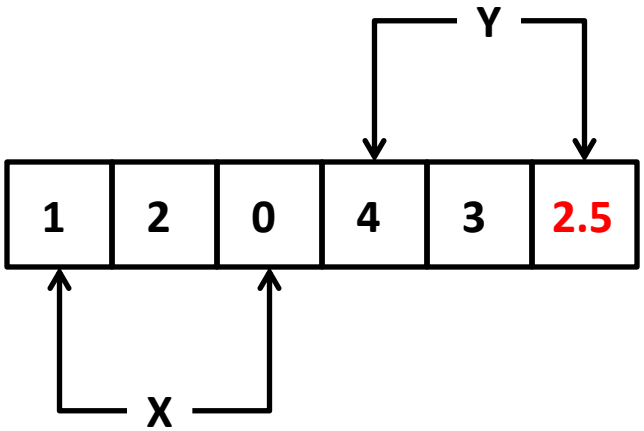
Partition

- X starts off empty
- Y is everything
- Take pivot = 3
- Iterate through our list, looking for things in Y that should be in X



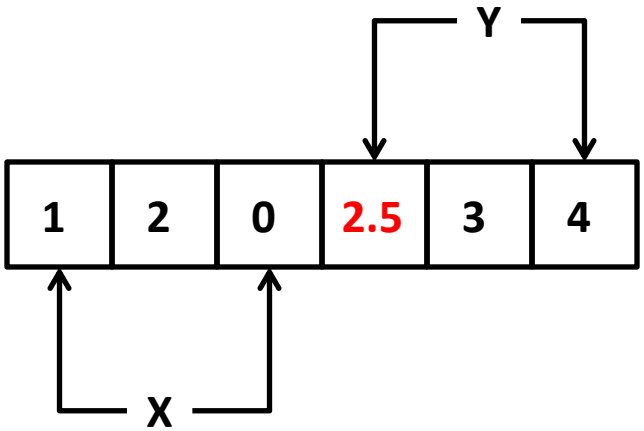
Partition

- X starts off empty
- Y is everything
- Take pivot = 3
- Iterate through our list, looking for things in Y that should be in X



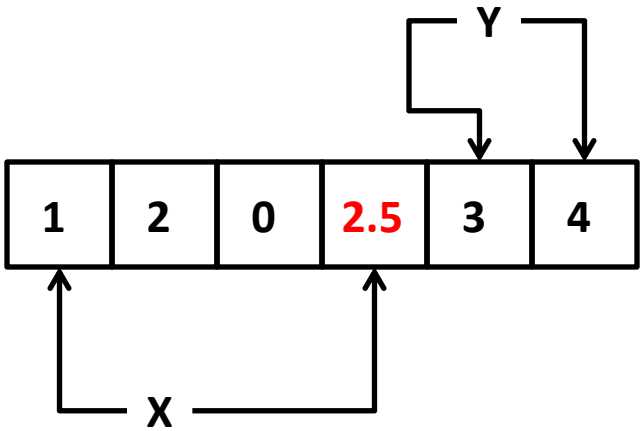
Partition

- X starts off empty
- Y is everything
- Take pivot = 3
- Iterate through our list, looking for things in Y that should be in X



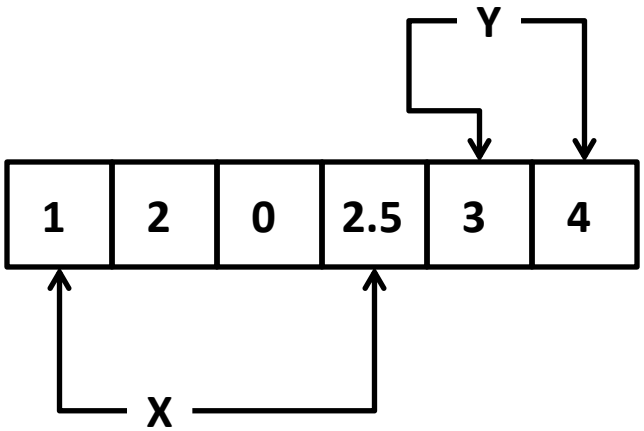
Partition

- X starts off empty
- Y is everything
- Take pivot = 3
- Iterate through our list, looking for things in Y that should be in X



Partition

- Done



Quicksort

- Given a list A[0:n], sort it
- Idea is to use partition
- Use A[n-1] as our pivot, and partition A[0:n-1]
- Let i be the value returned by partition
- Then A[0:i] is less than pivot, and A[i:n-1] is greater than or equal to pivot
- Swap A[i] with A[n-1], and recursively sort A[0:i] and A[i+1:n]

Quicksort

- The purpose for the swap was so that our recursive calls are always made on a *smaller* list (in the event that len(X) == 0)
- What is the runtime of this algorithm?

Quicksort Runtime

- Best case: partition always splits the list into two equal sized chunks
- Then we essentially get the same runtime as mergesort (O(n log n)), as the recurrence of the runtime is the same
- What about the worst case?

Quicksort Runtime

- In the worst case, the pivot was the the smallest (or largest) element in the list, and we recurse on something only one element smaller, giving us a quadratic runtime
- Fact: the average runtime is O(n log n)
- Conceptually, this is similar to why the average height of a BST is O(log n)

Quickselect

- Problem related to sorting
- Given a list of items, return the kth smallest item
- Ideas?

Quickselect

- Insert everything into a heap, extract_min k times
  - Runtime?
- Alternatively, we can modify quicksort so it does not sort the chunk of the list which does not contain the kth element…
  - Runtime?