Quicksort Algorithm For Arrays, Reading

Quicksort works for both arrays and linked lists. The algorithm is basically the same either way. Here it is for arrays.

It works like this -- first **choose a value** by it's *location*, usually **in the middle** of an unsorted array. Do so without regard for it's actual value, high, low, or in between.

17 21 4 9 **16** 1 5 1 50

Then with compares and swaps, rearrange the remaining values so that all those to the left of the chosen one are less-than-or-equal to it, and all those to its right are greater-than-or-equal. When you're done, the chosen value is now in its right place! And if you're lucky, the number of less-thans equals the number of greater-thans (but don't count on it!). This takes a number of operations proportional to "n", the size of the array.

1 5 4 9 1 **16** 21 17 50

Then repeat the process for the two sides of the array -- the ones to the left of the chosen value and the ones to the right:

1 5 **4** 9 1 16 21 **17** 50

Each side take about half the number of operations to rearrange, but since there are two to do, this step is also proportional to "n":

1 1 **4** 9 5 16 **17** 21 50

Each time you do this, the chosen value gets shifted into its proper, sorted location.

To reason out the big oh, note that **each step is O(n)**, because each time you cut a group in half and have half as many operations to do to position it's chosen middle value, you have twice as many groups! Step 1 has 1 group with n operations. Step 2 has 2 groups with n/2 operations each. Step 3 has 4 groups with n/4 operations each, and so on.

To get the overall big oh, it's the O(n) per step, times the **number of steps**. As you can probably see, doubling the number of values adds just one more step -- that's **O(log n)**, as we saw in previous modules for binary search and priority queue enqueue and dequeue. So that's where **O(n log n)** comes from.

Here's the algorithm for arrays. In it, the value we're calling the "chosen one" actually has a name -- it's the "pivot". The solution uses a "to-do" list to track the various simultaneous sorting of sections of the larger array.

```
define a struct with two ints to store a start index and an end index
create a "to-do" stack of int-int objects using the struct
create an int-int object with start=0 and end=n, and push it onto stack
while the stack is not empty
  create and set start index = stack top's start
  create and set end index = stack top's end
  pop the top of the stack
  if start index >= end index, "continue;"
  calculate the pivot's index = (start + end - 1) / 2

  create and set a left index = start
  create and set a right index = end - 1
  start a loop

    while left index < pivot && value at left index <= value at pivot
      increment left index

    while pivot < right index && value at pivot <= value at right index
      decrement right index
```

```
if left index == right index
  create an int-int object with start=start index, end=pivot; push
  create an int-int object with start=pivot+1, end=end index; push
  break out of the loop
else if left index == pivot
  swap value at right index with value at pivot
  set pivot index to right index
  increment left index
else if right index == pivot
  swap value at left index with value at pivot
  set pivot index to left index
  decrement right index
else
  swap value at left index with value at right index
  increment left index
  decrement right index
```

Watch This! (https://www.youtube.com/watch?v=vxENKIcs2Tw)

(https://www.youtube.com/watch?v=vxENKIcs2Tw)