Worked QuickSort Example from Slides

Original array:

0	1	2	3	4	5	6	7
7	**1**	**23**	**5**	**2**	**65**	**3**	**4**

To start:

quicksort(myData, 0, myData.length-1)

// FIRST LAYER OF CALLS

low = 0
high = 7

if low < high, continue with algorithm, otherwise do nothing -> low == 0, high == 7, therefore continue

<< PARTITION STEP STARTS >>

partition the array:

partition(data, low, high) -> partition(data, 0, 7)

set pivot = data[high] -> pivot = data[7] -> pivot = 4

set unpartitionedIndex = low -> unpartitionedIndex = 0

for each element in array between low (0) and high (7), do:

check if current element < pivot

if it is, swap element in current position with element in unpartitionedIndex

Increase unpartitionedIndex by 1

Once array has been looped through, swap element in unpartitionedIndex with element in high (the pivot)

0	1	2	3	4	5	6	7
7	**1**	**23**	**5**	**2**	**65**	**3**	**4**

Compare current element (7) with pivot (4) -> 7 > 4, no change needed

0	1	2	3	4	5	6	7
7	**1**	**23**	**5**	**2**	**65**	**3**	**4**

Compare current element (1) with pivot (4) -> 1 < 4, therefore swap needed:

After swap:

0	1	2	3	4	5	6	7
1	**7**	**23**	**5**	**2**	**65**	**3**	**4**

1 and 7 have swapped places. Now we have a partitioned element, we need to update the partition index

unpartitionedIndex++ -> unpartitionedIndex = 1

0	1	2	3	4	5	6	7
1	7	23	5	2	65	3	4

Compare current element (23) with pivot (4) -> 23 > 4, no change needed

0	1	2	3	4	5	6	7
1	7	23	5	2	65	3	4

Compare current element (5) with pivot (4) -> 5 > 4, no change needed

0	1	2	3	4	5	6	7
1	7	23	5	2	65	3	4

Compare current element (2) with pivot (4) -> 2 < 4, therefore swap needed:

After swap:

0	1	2	3	4	5	6	7
1	2	23	5	7	65	3	4

2 and 7 have swapped places. Now we have another partitioned element, we need to update the partition index

unpartitionedIndex++ -> unpartitionedIndex = 2

0	1	2	3	4	5	6	7
1	2	23	5	7	65	3	4

Compare current element (65) with pivot (4) -> 65 > 4, no change needed

0	1	2	3	4	5	6	7
1	2	23	5	7	65	3	4

Compare current element (3) with pivot (4) -> 3 < 4, therefore swap needed:

After swap:

0	1	2	3	4	5	6	7
1	2	3	5	7	65	23	4

3 and 23 have swapped places. Now we have another partitioned element, we need to update the partition index

unpartitionedIndex++ -> unpartitionedIndex = 3

At this stage, we have iterated through the entire array (for loop has completed)

Update the array again to move the pivot value to the end of the partitioned data:

I.e. swap the element in data[high] with the element in data[unpartitionedIndex]

0	1	2	3	4	5	6	7
1	2	3	4	7	65	23	5

4 and 5 have swapped places.

The data partitioning has completed. Finally, return the unpartitionedIndex value (as this points to where the pivot now resides in the array)

<< PARTITION STEP COMPLETE >>

~~int pivotIndex = partition(data, low, high);~~ // This step is done, now we have to sort the parts
quicksort(data, low, pivotIndex-1) -> quicksort(data, 0, (3-1))
quicksort(data, pivotIndex+1, high) -> quicksort(data, (3+1), 7)

quicksort(data, 0, (3-1)):

low = 0
high = 2

// SECOND LAYER OF CALLS

if low < high, continue with algorithm, otherwise do nothing -> low == 0, high == 2, therefore continue

Partitioning the array:

<< PARTITION STEP STARTS >>

Pivot = data[2] (high)
unpartitionedIndex = 0 (low)

Current active partition

0	1	2	3	4	5	6	7
1	**2**	**3**	**4**	**7**	**65**	**23**	**5**

Compare current element (1) with pivot (3) -> 1 < 3, therefore swap needed

After swap:

0	1	2	3	4	5	6	7
1	**2**	**3**	**4**	**7**	**65**	**23**	**5**

1 & 1 have swapped places. Now we have a partitioned element, we need to update the partition index

unpartitionedIndex++ -> unpartitionedIndex = 1

0	1	2	3	4	5	6	7
1	**2**	**3**	**4**	**7**	**65**	**23**	**5**

Compare current element (2) with pivot (3) -> 2 < 3, therefore swap needed

After swap:

0	1	2	3	4	5	6	7
1	**2**	**3**	**4**	**7**	**65**	**23**	**5**

2 & 2 have swapped places. Now we have another partitioned element, we need to update the partition index

unpartitionedIndex++ -> unpartitionedIndex = 2

At this stage, we have iterated through the entire array section (for loop has completed)

Update the array again to move the pivot value to the end of the partitioned data:

I.e. swap the element in data[high] with the element in data[unpartitionedIndex]

0	1	2	3	4	5	6	7
1	**2**	**3**	**4**	**7**	**65**	**23**	**5**

3 & 3 have swapped places.

The data partitioning has completed. Finally, return the unpartitionedIndex value (as this points to where the pivot now resides in this section of the array)

<< PARTITION STEP COMPLETE>>

~~int pivotIndex = partition(data, low, high);~~ // This step is done, now we have to sort the parts
quicksort(data, low, pivotIndex-1) -> quicksort(data, 0, (2-1))
quicksort(data, pivotIndex+1, high) -> quicksort(data, (2+1), 2)

quicksort(data, 0, (2-1)):

low = 0
high = 1

// THIRD LAYER OF CALLS

if low < high, continue with algorithm, otherwise do nothing -> low == 0, high == 1, therefore continue

Partitioning the array:

<< PARTITION STEP STARTS >>

Pivot = data[1] (high)
unpartitionedIndex = 0 (low)

Current active partition

0	1	2	3	4	5	6	7
1	**2**	**3**	**4**	**7**	**65**	**23**	**5**

Compare current element (1) with pivot (2) -> 1 < 2, therefore swap needed

After swap:

0	1	2	3	4	5	6	7
1	**2**	**3**	**4**	**7**	**65**	**23**	**5**

1 & 1 have swapped places. Now we have a partitioned element, we need to update the partition index

unpartitionedIndex++ -> unpartitionedIndex = 1

At this stage, we have iterated through the entire array section (for loop has completed)

Update the array again to move the pivot value to the end of the partitioned data:

I.e. swap the element in data[high] with the element in data[unpartitionedIndex]

0	1	2	3	4	5	6	7
1	**2**	**3**	**4**	**7**	**65**	**23**	**5**

2 & 2 have swapped places.

The data partitioning has completed. Finally, return the unpartitionedIndex value (as this points to where the pivot now resides in this section of the array)

<< PARTITION STEP COMPLETE>>

~~int pivotIndex = partition(data, low, high);~~ // This step is done, now we have to sort the parts
quicksort(data, low, pivotIndex-1) -> quicksort(data, 0, (1-1))
quicksort(data, pivotIndex+1, high) -> quicksort(data, (1+1), 1)

quicksort(data, 0, (1-1)):

low = 0
high = 0

// FOURTH LAYER OF CALLS

if low < high, continue with algorithm, otherwise do nothing -> low == 0, high == 0, therefore recursion stops and method terminates

// BACK UP TO THIRD LAYER OF CALLS

~~int pivotIndex = partition(data, low, high);~~ // This step is done, now we have to sort the parts
~~quicksort(data, low, pivotIndex-1)~~ -> quicksort(data, 0, (1-1)) // This step is now done, start of this section is sorted
quicksort(data, pivotIndex+1, high) -> quicksort(data, (1+1), 1)

quicksort(data, (1+1), 1):

low = 2
high = 1

// FOURTH LAYER OF CALLS

if low < high, continue with algorithm, otherwise do nothing -> low == 2, high == 1, therefore recursion stops and method terminates

// BACK UP TO THIRD LAYER OF CALLS

~~int pivotIndex = partition(data, low, high);~~ // COMPLETED
~~quicksort(data, low, pivotIndex-1)~~ -> quicksort(data, 0, (1-1)) // COMPLETED
~~quicksort(data, pivotIndex+1, high)~~ -> quicksort(data, (1+1), 1) // This step is now done, end of this section is sorted

// BACK UP TO SECOND LAYER OF CALLS

~~int pivotIndex = partition(data, low, high);~~ // COMPLETED
~~quicksort(data, low, pivotIndex-1)~~ -> quicksort(data, 0, (2-1)) // This step is now done, start of this section is sorted
quicksort(data, pivotIndex+1, high) -> quicksort(data, (2+1), 2)

quicksort(data, (2+1), 2):

low = 3
high = 2

// THIRD LAYER OF CALLS

if low < high, continue with algorithm, otherwise do nothing -> low == 3, high == 2, therefore recursion stops and method terminates

// BACK UP TO SECOND LAYER OF CALLS

~~int pivotIndex = partition(data, low, high);~~ // COMPLETED
~~quicksort(data, low, pivotIndex-1)~~ -> quicksort(data, 0, (2-1)) // COMPLETED
~~quicksort(data, pivotIndex+1, high)~~ -> quicksort(data, (2+1), 2) // This step is now done, end of this section is sorted
// BACK UP TO FIRST LAYER OF CALLS

At this stage, the entire first subsection of the array has been fully sorted, plus the original pivot element (4)

0	1	2	3	4	5	6	7
1	**2**	**3**	**4**	**7**	**65**	**23**	**5**

~~int pivotIndex = partition(data, low, high);~~ // COMPLETED
~~quicksort(data, low, pivotIndex-1)~~ -> quicksort(data, 0, (3-1)) // Left subsection is fully sorted
quicksort(data, pivotIndex+1, high) -> quicksort(data, (3+1), 7)

quicksort(data, (3+1), 7):

low = 4
high = 7

// SECOND LAYER OF CALLS

if low < high, continue with algorithm, otherwise do nothing -> low == 4, high == 7, therefore continue

Partitioning the array:

<< PARTITION STEP STARTS >>

Pivot = data[7] (high)
unpartitionedIndex = 2 (low)

Current active partition

0	1	2	3	4	5	6	7
1	**2**	**3**	**4**	**7**	**65**	**23**	**5**

Compare current element (7) with pivot (5) -> 7 > 5, no change needed

0	1	2	3	4	5	6	7
1	**2**	**3**	**4**	**7**	**65**	**23**	**5**

Compare current element (65) with pivot (5) -> 65 > 5, no change needed

0	1	2	3	4	5	6	7
1	**2**	**3**	**4**	**7**	**65**	**23**	**5**

Compare current element (23) with pivot (5) -> 23 > 5, no change needed

At this stage, we have iterated through the entire array section (for loop has completed)

Update the array again to move the pivot value to the end of the partitioned data:

I.e. swap the element in data[high] with the element in data[unpartitionedIndex]

0	1	2	3	4	5	6	7
1	**2**	**3**	**4**	**5**	**65**	**23**	**7**

7 & 5 have swapped places.

Note: unpartitionedIndex did not change at all during the course of this loop

The data partitioning has completed. Finally, return the unpartitionedIndex value (as this points to where the pivot now resides in this section of the array)

<< PARTITION STEP COMPLETE>>

~~int pivotIndex = partition(data, low, high);~~ // This step is done, now we have to sort the parts
quicksort(data, low, pivotIndex-1) -> quicksort(data, 4, (4-1))
quicksort(data, pivotIndex+1, high) -> quicksort(data, (4+1), 7)

quicksort(data, 4, (4-1)):

low = 4
high = 3

// THIRD LAYER OF CALLS

if low < high, continue with algorithm, otherwise do nothing -> low == 4, high == 3, therefore recursion stops and method terminates

// BACK UP TO SECOND LAYER OF CALLS

~~int pivotIndex = partition(data, low, high);~~ // COMPLETED
~~quicksort(data, low, pivotIndex-1)~~ -> quicksort(data, 4, (4-1)) // Front of this subsection now sorted
quicksort(data, pivotIndex+1, high) -> quicksort(data, (4+1), 7)

quicksort(data, (4+1), 7):

low = 5
high = 7

// THIRD LAYER OF CALLS

if low < high, continue with algorithm, otherwise do nothing -> low == 5, high == 7, therefore continue

Partitioning the array:

<< PARTITION STEP STARTS >>

Pivot = data[7] (high)
unpartitionedIndex = 5 (low)

Current active partition

0	1	2	3	4	5	6	7
1	**2**	**3**	**4**	**5**	**65**	**23**	**7**

Compare current element (65) with pivot (7) -> 65 > 7, no change needed

0	1	2	3	4	5	6	7
1	**2**	**3**	**4**	**5**	**65**	**23**	**7**

Compare current element (23) with pivot (7) -> 23 > 7, no change needed

At this stage, we have iterated through the entire array section (for loop has completed)

Update the array again to move the pivot value to the end of the partitioned data:

I.e. swap the element in data[high] with the element in data[unpartitionedIndex]

0	1	2	3	4	5	6	7
1	**2**	**3**	**4**	**5**	**7**	**23**	**65**

7 & 65 have swapped places.

Note: unpartitionedIndex did not change at all during the course of this loop

The data partitioning has completed. Finally, return the unpartitionedIndex value (as this points to where the pivot now resides in this section of the array)

<< PARTITION STEP COMPLETE>>

~~int pivotIndex = partition(data, low, high);~~ // This step is done, now we have to sort the parts
quicksort(data, low, pivotIndex-1) -> quicksort(data, 5, (5-1))
quicksort(data, pivotIndex+1, high) -> quicksort(data, (5+1), 7)

quicksort(data, 5, (5-1)):

low = 5
high = 4

// FOURTH LAYER OF CALLS

if low < high, continue with algorithm, otherwise do nothing -> low == 5, high == 4, therefore recursion stops and method terminates

// BACK UP TO THIRD LAYER OF CALLS

~~int pivotIndex = partition(data, low, high);~~ // COMPLETED
~~quicksort(data, low, pivotIndex-1)~~ -> quicksort(data, 5, (5-1)) // Front of this subsection now sorted
quicksort(data, pivotIndex+1, high) -> quicksort(data, (5+1), 7)

quicksort(data, (5+1), 7):

low = 6
high = 7

// FOURTH LAYER OF CALLS

if low < high, continue with algorithm, otherwise do nothing -> low == 6, high == 7, therefore continue

Partitioning the array:

<< PARTITION STEP STARTS >>

Pivot = data[7] (high)
unpartitionedIndex = 6 (low)

Current active partition

0	1	2	3	4	5	6	7
1	2	3	4	5	7	23	65

Compare current element (23) with pivot (65) -> 23 < 65, therefore swap required

0	1	2	3	4	5	6	7
1	2	3	4	5	7	23	65

23 & 23 have swapped places. Now we have a partitioned element, we need to update the partition index

unpartitionedIndex++ -> unpartitionedIndex = 7

At this stage, we have iterated through the entire array section (for loop has completed)

Update the array again to move the pivot value to the end of the partitioned data:

I.e. swap the element in data[high] with the element in data[unpartitionedIndex]

0	1	2	3	4	5	6	7
1	2	3	4	5	7	23	65

65 & 65 have swapped places.

The data partitioning has completed. Finally, return the unpartitionedIndex value (as this points to where the pivot now resides in this section of the array)

<< PARTITION STEP COMPLETE>>

~~int pivotIndex = partition(data, low, high);~~ // This step is done, now we have to sort the parts
quicksort(data, low, pivotIndex-1) -> quicksort(data, 6, (7-1))
quicksort(data, pivotIndex+1, high) -> quicksort(data, (7+1), 7)

quicksort(data, 6, (7-1)):

low = 6
high = 6

// FIFTH LAYER OF CALLS

if low < high, continue with algorithm, otherwise do nothing -> low == 6, high == 6, therefore recursion stops and method terminates

// BACK UP TO FOURTH LAYER OF CALLS

~~int pivotIndex = partition(data, low, high);~~ // COMPLETED
~~quicksort(data, low, pivotIndex-1)~~ -> quicksort(data, 6, (7-1)) // Front of this subsection now sorted
quicksort(data, pivotIndex+1, high) -> quicksort(data, (7+1), 7)

quicksort(data, (7+1), 7):

low = 8
high = 7

// FIFTH LAYER OF CALLS

if low < high, continue with algorithm, otherwise do nothing -> low == 8, high == 7, therefore recursion stops and method terminates

// BACK UP TO FOURTH LAYER OF CALLS

Structure at this level:

0	1	2	3	4	5	6	7
1	**2**	**3**	**4**	**5**	**7**	**23**	**65**

~~int pivotIndex = partition(data, low, high);~~ // COMPLETED
~~quicksort(data, low, pivotIndex-1)~~ -> quicksort(data, 6, (7-1)) // COMPLETED
~~quicksort(data, pivotIndex+1, high)~~ -> quicksort(data, (7+1), 7) // This step is now done, end of this section is sorted

// BACK UP TO THIRD LAYER OF CALLS
Structure at this level:

0	1	2	3	4	5	6	7
1	**2**	**3**	**4**	**5**	**7**	**23**	**65**

~~int pivotIndex = partition(data, low, high);~~ // COMPLETED
~~quicksort(data, low, pivotIndex-1)~~ -> quicksort(data, 5, (5-1)) // COMPLETED
~~quicksort(data, pivotIndex+1, high)~~ -> quicksort(data, (5+1), 7) // This step is now done, end of this section is sorted

// BACK UP TO SECOND LAYER OF CALLS
Structure at this level:

0	1	2	3	4	5	6	7
1	**2**	**3**	**4**	**5**	**7**	**23**	**65**

~~int pivotIndex = partition(data, low, high);~~ // COMPLETED
~~quicksort(data, low, pivotIndex-1)~~ -> quicksort(data, 4, (4-1)) // COMPLETED
~~quicksort(data, pivotIndex+1, high)~~ -> quicksort(data, (4+1), 7) // This step is now done, end of this section is sorted

// BACK UP TO FIRST LAYER OF CALLS

Structure at this level:

0	1	2	3	4	5	6	7
1	**2**	**3**	**4**	**5**	**7**	**23**	**65**

~~int pivotIndex = partition(data, low, high);~~ // COMPLETED
~~quicksort(data, low, pivotIndex-1)~~ -> quicksort(data, 0, (3-1)) // COMPLETED
quicksort(data, pivotIndex+1, high) -> quicksort(data, (3+1), 7) // This step is now done, tail of array is now sorted

FINAL SORTED STRUCTURE:

0	1	2	3	4	5	6	7
1	**2**	**3**	**4**	**5**	**7**	**23**	**65**

Call Tree for Example: