CS 112: Data Structures

Sesh Venugopal

Quicksort Algorithm

Divide and Conquer

Mergesort and Quicksort both use what's called a "divide and conquer" technique

Divide and Conquer Algorithms

input: list *L_in* to be sorted
output: sorted list *L_out*

divide *L_in* into *L_left*, *L_right*
recursively sort *L_left* into *L_left_sorted*
recursively sort *L_right* into *L_right_sorted*
combine *L_left_sorted*, *L_right_sorted* into *L_out*)

The first step is to *divide* the input list, *L_in*, into two parts, *L_left* and *L_right*.

Each part is sorted recursively by applying the same divide-and-conquer strategy: the sorted parts are in *L_left_sorted* and *L_right_sorted*, respectively.

The last step is to combine these two sorted parts to produce the sorted list, *L_out*.

In Mergesort, the "divide" step is trivial – just divide the array in two halves. All the work is done in "combine", where sorted subarrays are merged

Divide and Conquer in Quicksort

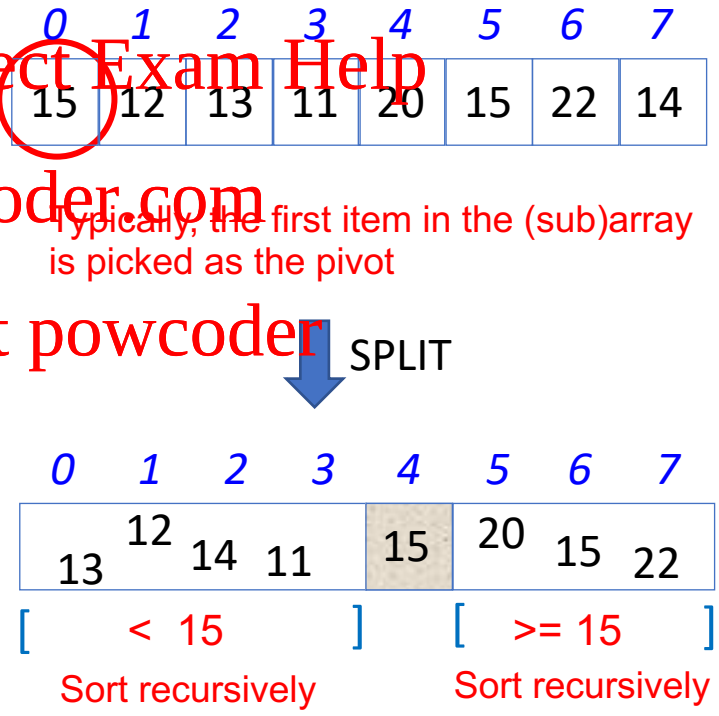
In Quicksort, the divide and conquer process works in exactly the opposite way. As in, all the work is done in the “divide” step, and nothing at all is done in the ”combine” step.

Algorithm quicksort(A, left, right)

input: subarray $A[left \ldots right]$
output: sorted subarray $A[left \ldots right]$

$splitPoint \leftarrow$ **split**$(A, left, right)$;
quicksort$(A, left, splitPoint - 1)$;
quicksort$(A, splitPoint + 1, right)$;

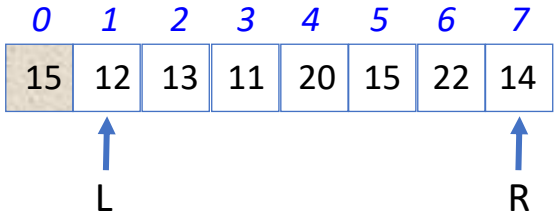
The *split* process selects a *pivot* entry, say x, in the given sublist and rearranges the entries of the sublist in such a way that all the entries less than x are to its left, and all other entries (greater than or equal to x) are to its right.



After the split, the pivot is in its correct sorted place

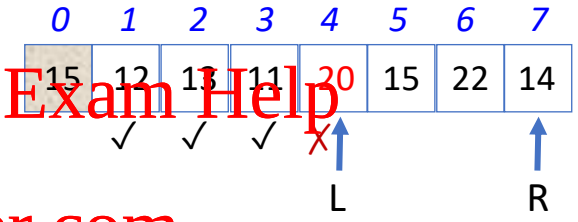
Split

Pivot is the first value, 15
Start with L at the next index after the pivot, and R at the last index

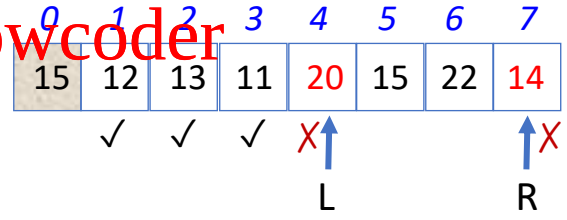


Iteration 1

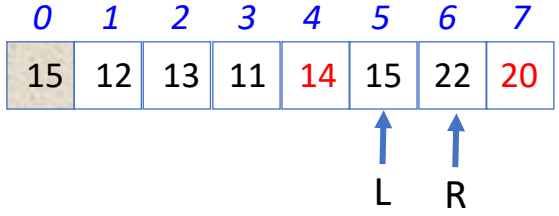
If A[L] < pivot, then advance L to next index, otherwise stop – here L advances all the way to 20, where it stops because 20 is not < than 15



If A[R] >= pivot, then advance R to previous index, otherwise stop – here R does not advance at all since 14 is not >= 15

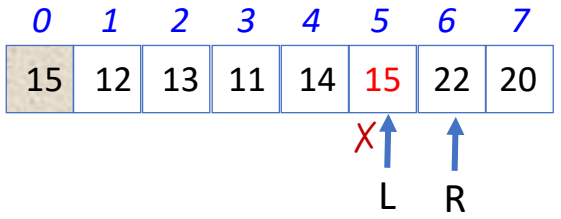


Swap the values at L and R, then advance L up by 1 and R down by 1, to prepare for next iteration

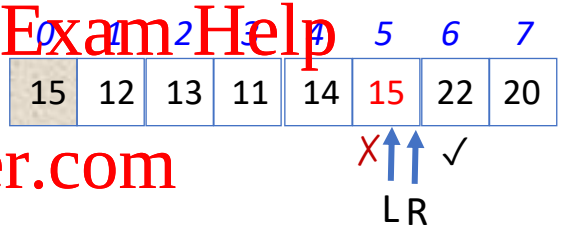


Iteration 2

If A[L] < pivot, then advance L to next index, otherwise stop – here L does not advance at all since 15 is not < 15



If A[R] >= pivot, then advance R to previous index, otherwise stop – here R advances to 15

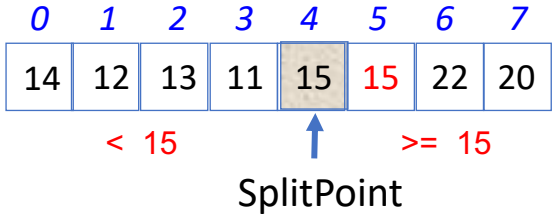


Normally it would have also compared 15, but because it has already been compared via L, R stops without comparing 15 against the pivot.

So the condition to advance R is modified to this:
If R > L and A[R] >= pivot, then R--

At this point, both L and R have finished moving as far as they could, because R is not > L

So we need wrap up. This is done by swapping A[L-1] with the pivot:



Split Example 2

Pivot is 9

Iteration 1

left starts at item 3, right starts at item 12
left moves up to 14 and stops (14 not < 9),

right moves down to 5 and stops (5 not >= 9)
swap 14 with 5, then left++, right--

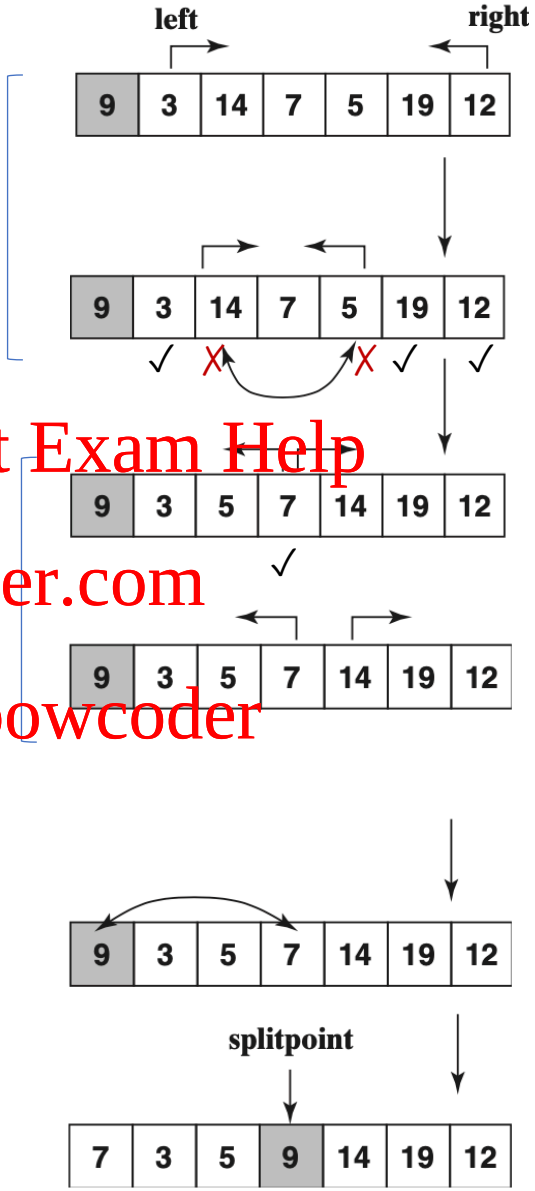
Iteration 2

left starts at item 7, right starts at item 7
left moves past 7, but stops at 14 without comparing 14
because left has moved past right

So the condition to advance L is modified to this:
If L <= R and A[L] < pivot, then L++

left and right have crossed over, so right does not move
at all (remember right only moves if right > left)

All items have been compared against pivot
(right is not > left)
Swap A[left-1] (7) with pivot (9)

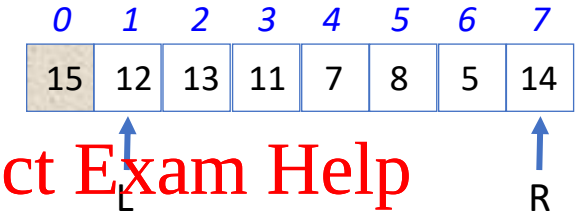


Split: Extreme Case 1 (All items < pivot)

Condition to advance L:
If L <= R && A[L] < pivot, then L++

Condition to advance R:
If R > L && A[R] >= pivot, then R--

Pivot is 15

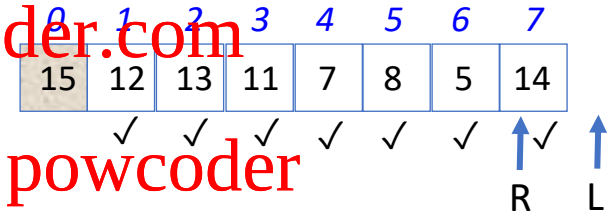


Iteration 1

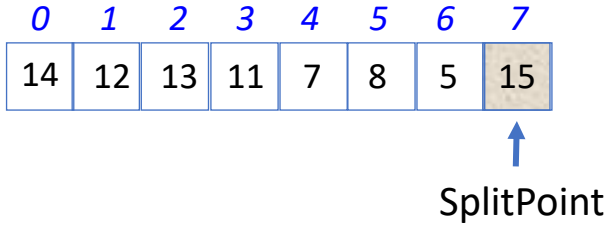
L starts at item 12, R starts at item 14

L moves all the way through the array and stops when it moves past R (and goes out of bound)

R does not move at all, since R is not > L



All items have been compared against pivot (R is not > L)
Swap A[L-1] (14) with pivot (15)

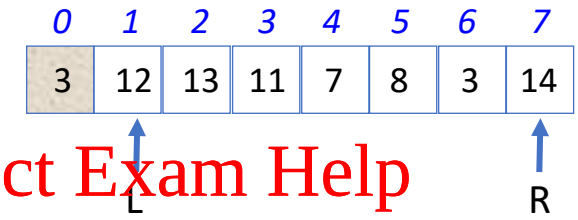


Split: Extreme Case 2 (All items >= pivot)

Condition to advance L:
If L <= R && A[L] < pivot, then L++

Condition to advance R:
If R > L && A[R] >= pivot, then R--

Pivot is 3

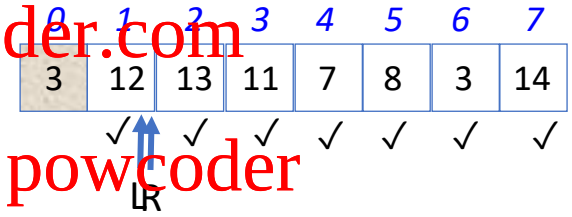


Iteration 1

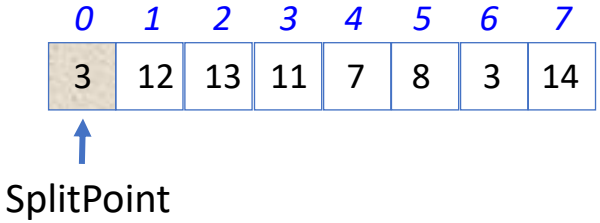
L starts at item12, R starts at item 14

L does not move at all (since 12 not < 3)

R moves all the way across. Past 13, it goes to item 12, but stops (without making a comparison) because R not > L



All items have been compared against pivot
(R is not > L)
Swap A[L-1] (3) with pivot (3), i.e. swap pivot with itself



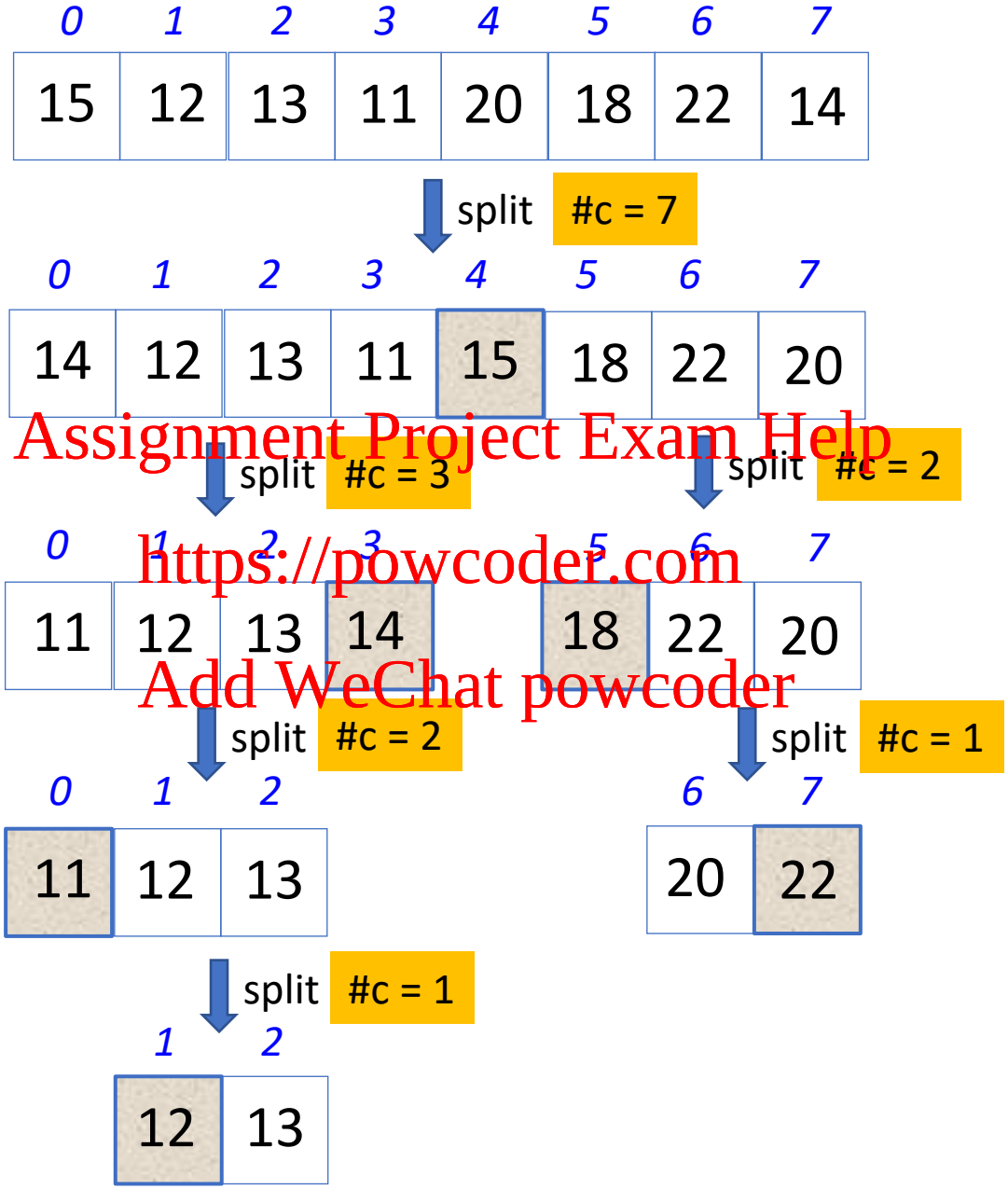
```
Algorithm split(A, lo, hi)
    pivot ← A[lo]
    left ← lo+1, right ← hi
    while (true) do
       while (left <= right) do
          if (A[left] < pivot) then
             left++
          else
             break
          endif
       endwhile

       while (right >= left) do
          if (A[right] < pivot) then
             break;
          else
             right-
          endif
       endwhile

       if (left >= right) then
          break
       endif

        swap(A[left],A[right])
       left++
       right-
    endwhile

    swap(A[left-1],A[lo])
    return left-1
```

Algorithm quicksort(A, left, right)

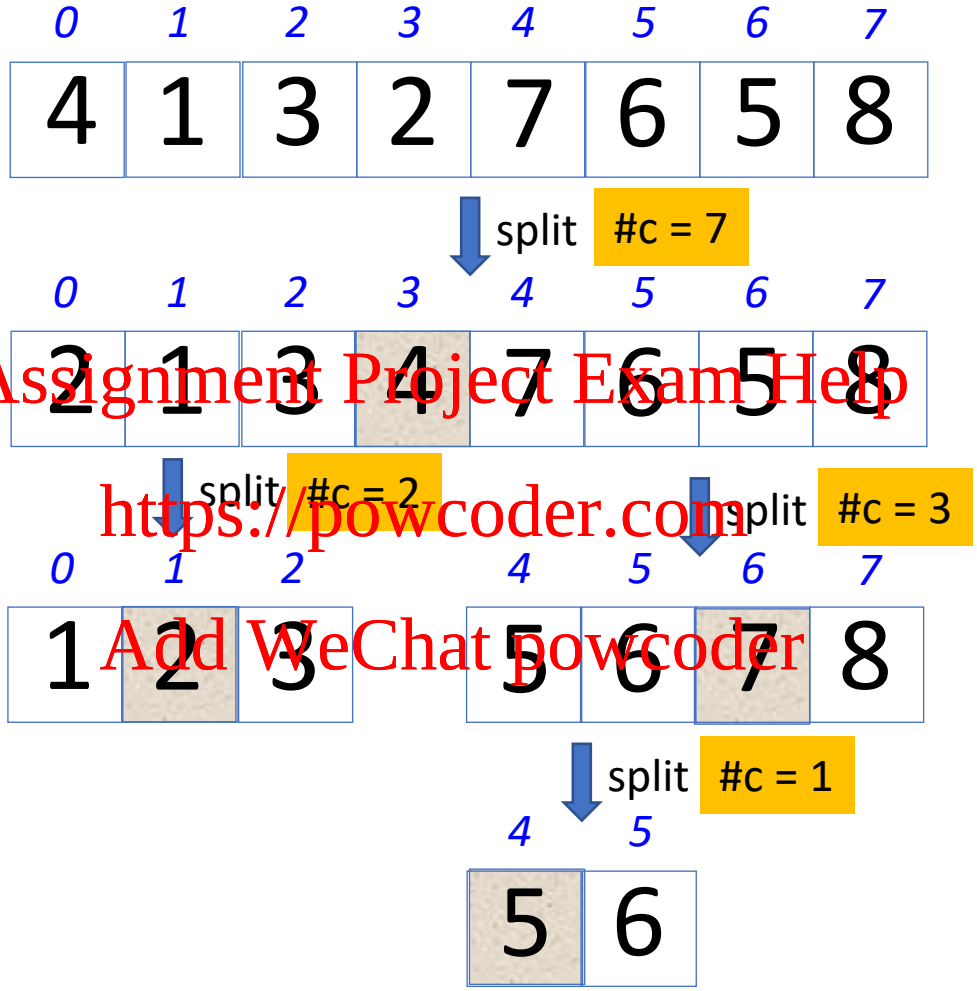
input: subarray $A[left \ldots right]$
output: sorted subarray $A[left \ldots right]$

$splitPoint \leftarrow$ **split**$(A, left, right)$
quicksort$(A, left, splitPoint - 1)$;
quicksort$(A, splitPoint + 1, right)$;

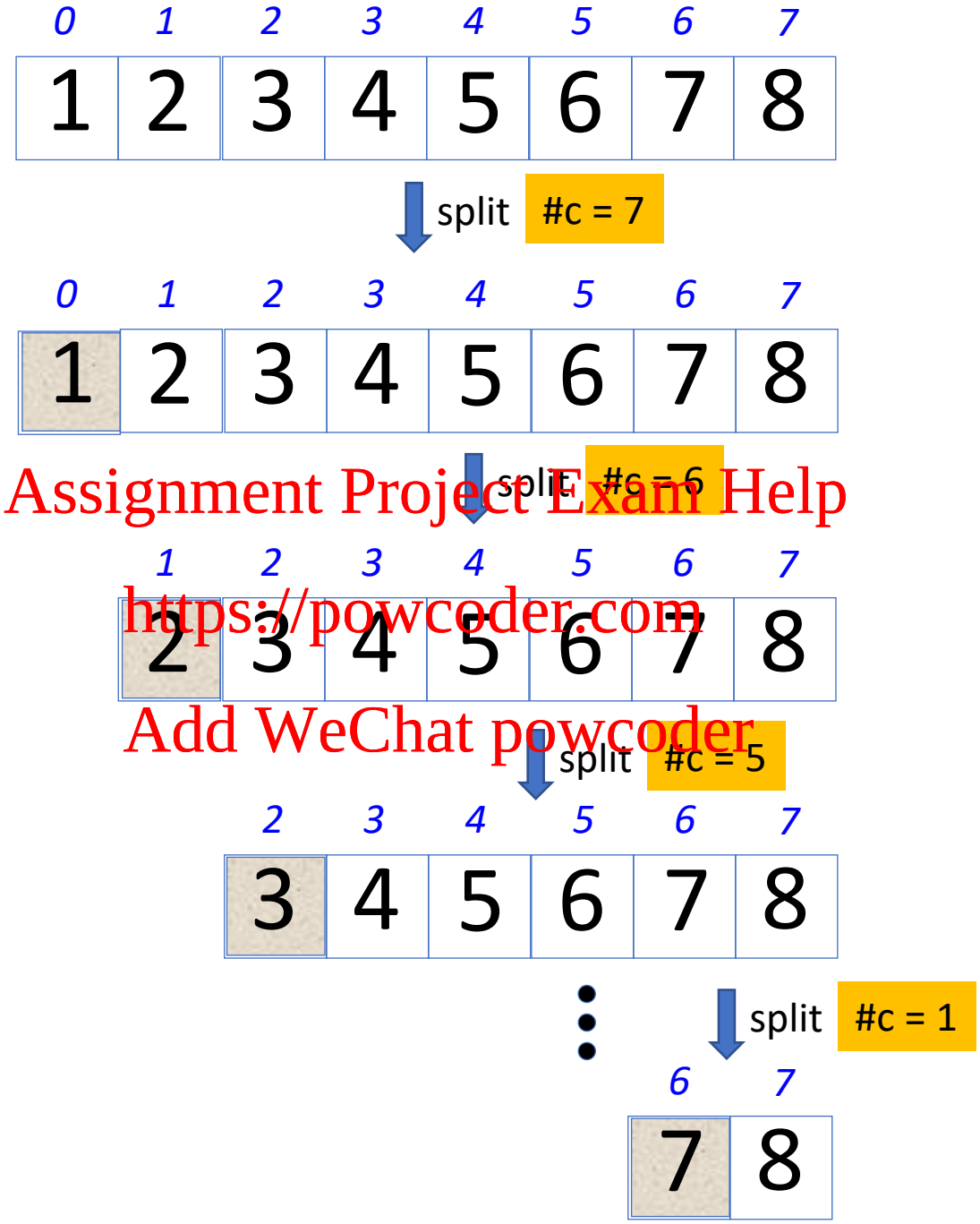
Recursion Tree Example 1



Recursion Tree Example 2



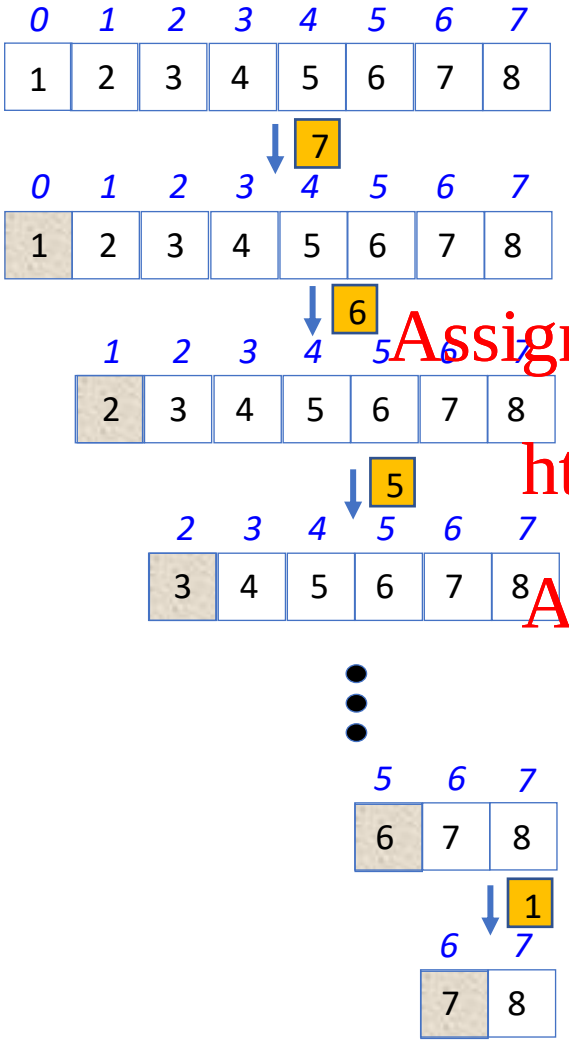
Recursion Tree Example 3



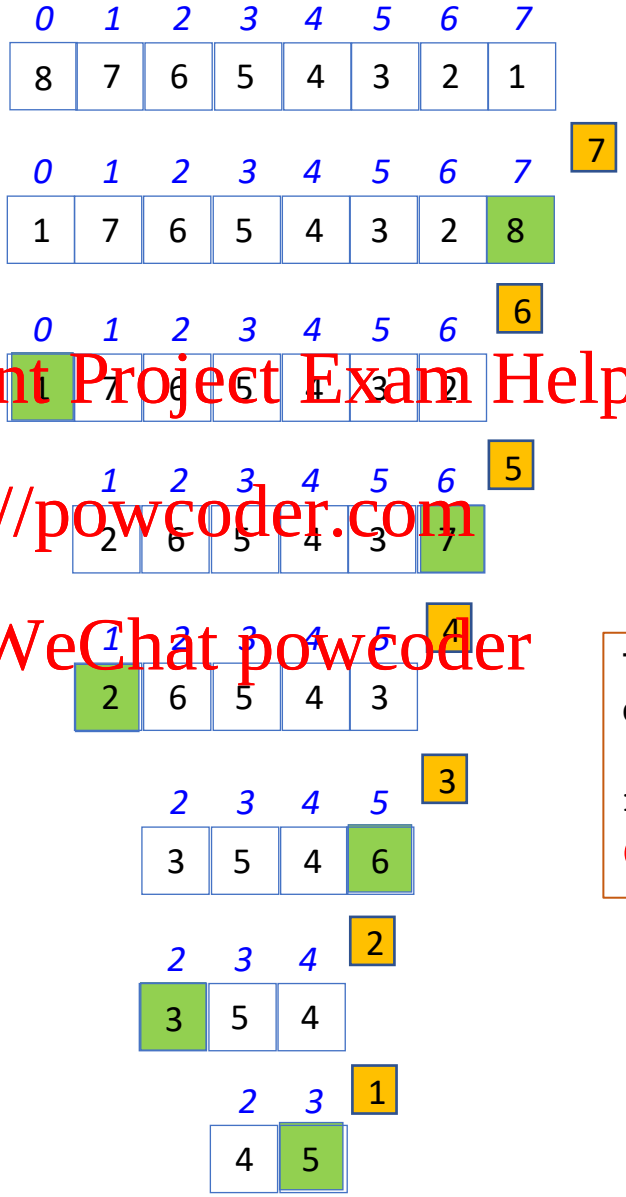
Quicksort Running Time

Worst case

Sorted Input

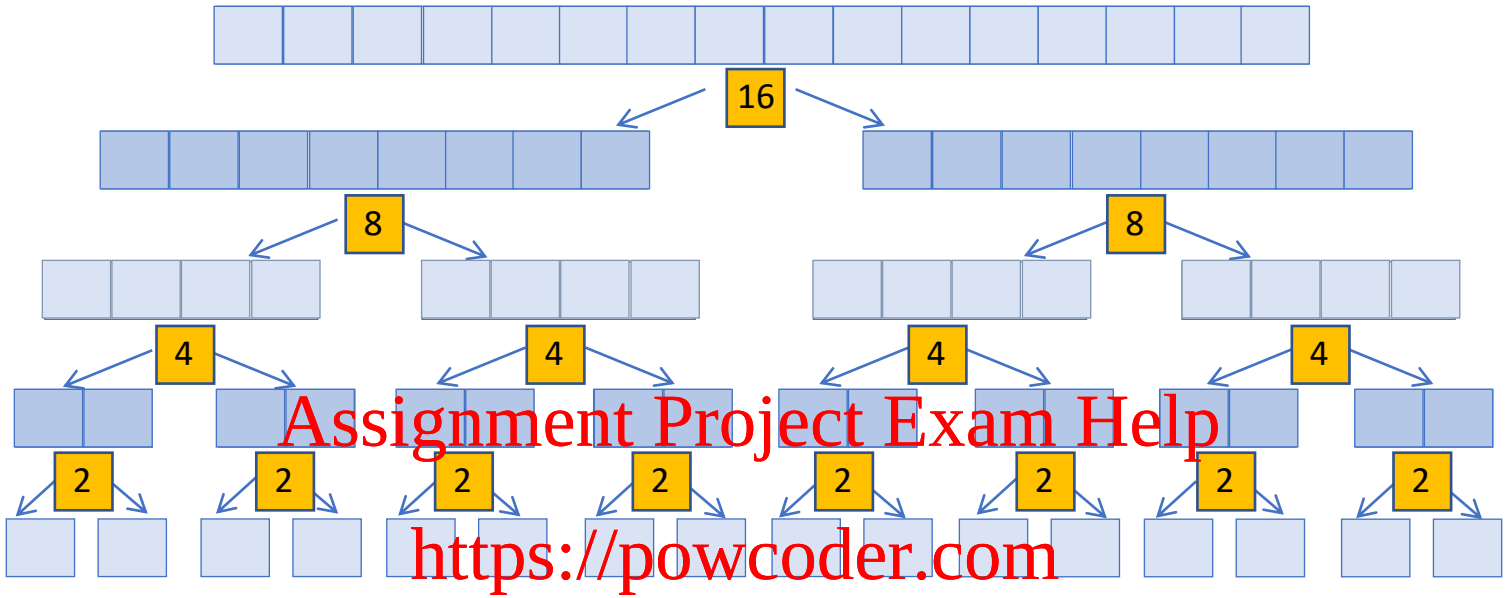


Reverse Sorted Input



Total number of comparisons =
$(n-1)+(n-2)+\ldots 2+1 =$
$n(n-1)/2 =$
$O(n^2)$

Best case



(n = 16, the number of comparisons are overcounted somewhat, for ease of analysis but the big O will be unchanged)

Total number of comparisons
= 16 + 16 + 16 + 16 =
= 16*4 (height)
= $16*\log_2(16)$

Which generalizes to
O(nlogn)

So why is Quicksort popular?

	Worst Case	Best Case	Average Case
Insertion Sort	$O(n^2)$	$O(n)$	$O(n^2)$
Quicksort	$O(n^2)$	$O(n \log n)$	$O(n \log n)$