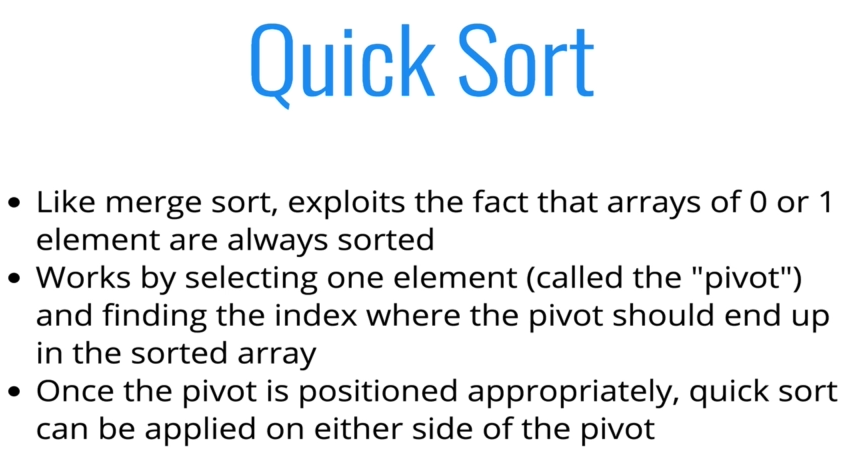
Chapter 6 : Part 6

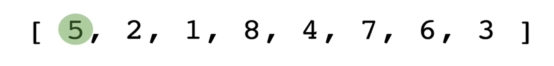
**Sorting Algorithms: Quick Sort**

**6.6.1 Quick Sort**

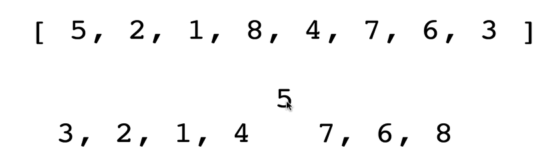
* It works using same assumption of Merge-Sort.
* Easiest to solve through recursion.
* We split the array until each sub-array has 1 or 0 elements (base case), assuming these sub-array of 1 or 0c elements are always sorted.
* Pivot: In case of quick sort we use a value as a ***pivot***. Pivot can be anywhere, generally we select this p***i***vot near the ***middle-index***.
* We move the values to Left/Right side according to this pivot. Note that, we are just moving the values, not sorting.



**6.6.2 How does it works**

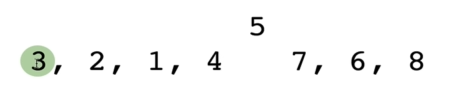


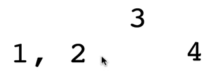
* All numbers less than 5 moved to left, and others that are greater than 5 are at right side of 5. In this case we move 5 to the position (index) of 4 (current position).



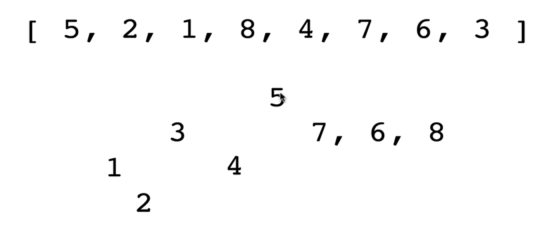
Now notice that, 5 is now at index-4, and already sorted.

* Basically, we repeat the process in the left and right portion of the array.

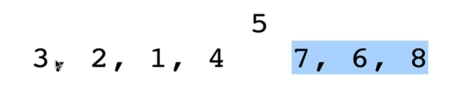




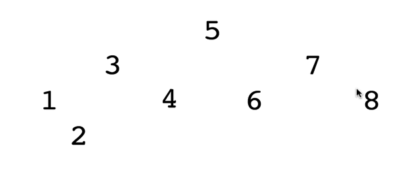
* We do the same for 1, 2;



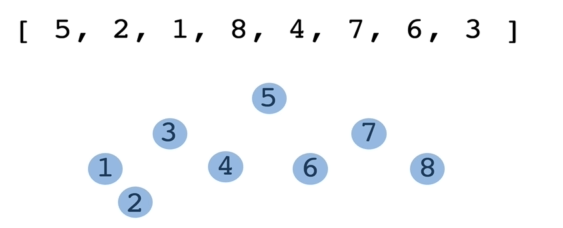
* Now notice that, the left side of the 5 is already sorted. We do the same for the right side.



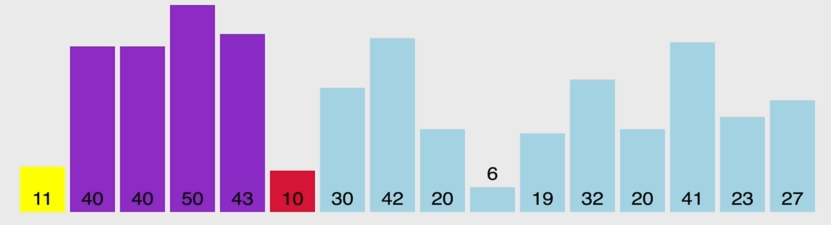
* Note that, we do this until we hit the "single element array" .



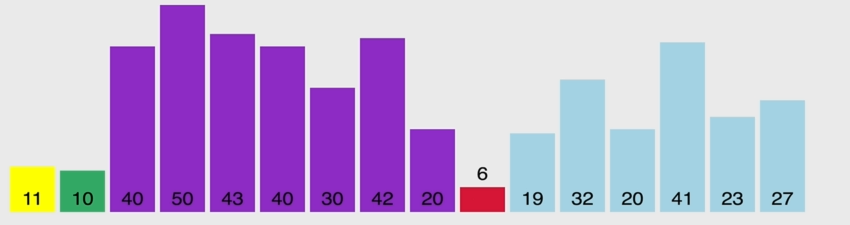
Finally we get the sorted array.



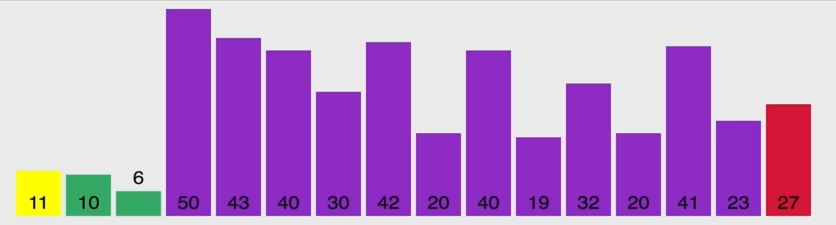
* Visualgo diagram:



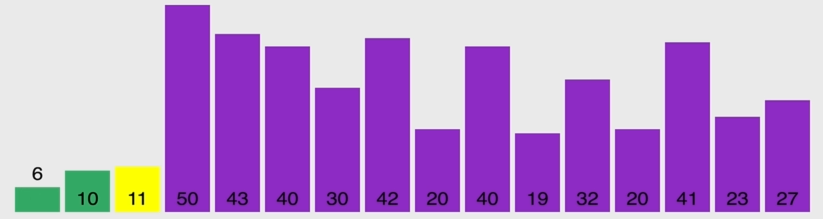
* Instead of swap the smaller values (11 & 10), to track the number of smaller values than the pivot we swap with the higher value next to pivot (i.e. 40 & 10).



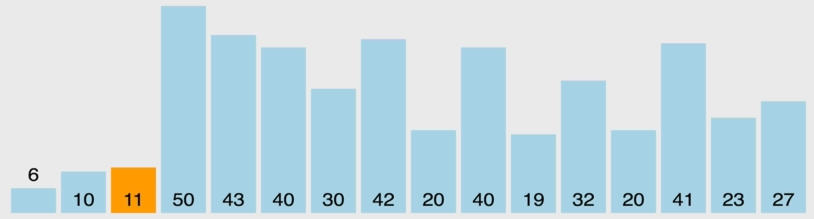
* We do same for the 6.

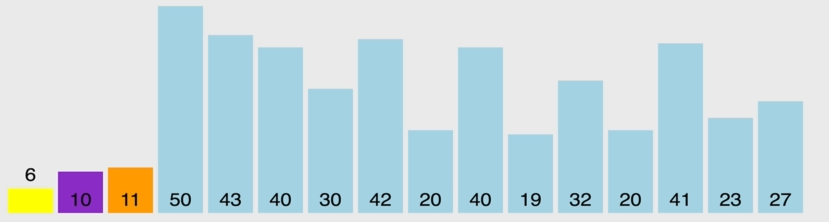


* After finding all the smaller values, we then move the pivot to its sorted-index (in this case index-2).

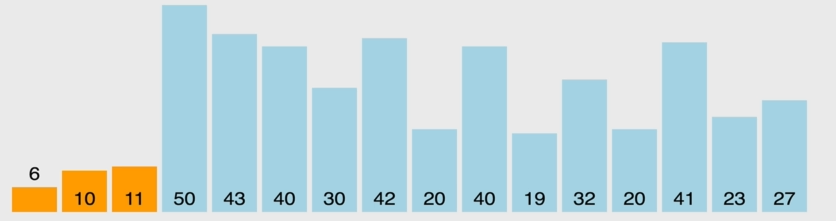


* Since 11 is already sorted, it is fixed now. We repeat the process for the left-side and then Right side.

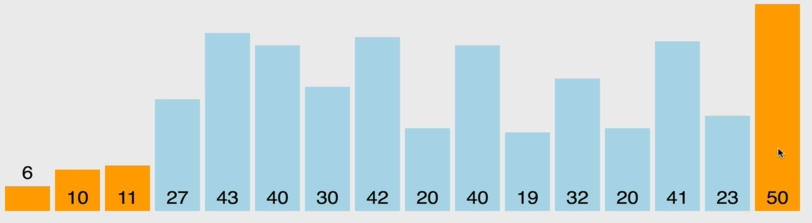


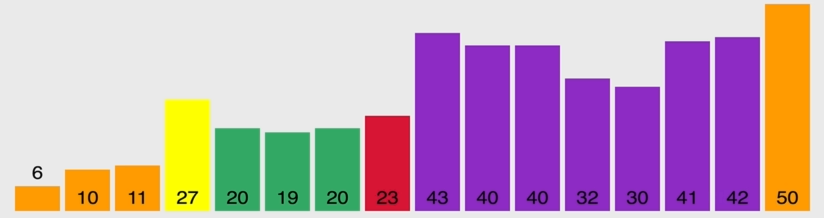


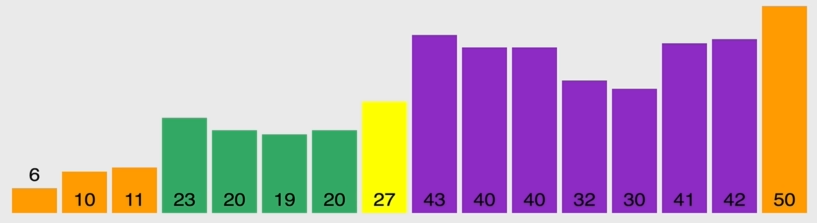
* Now the left side is sorted



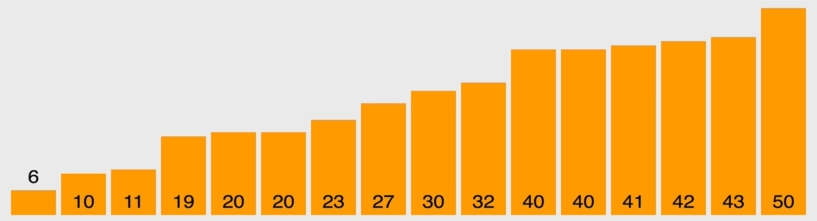
* Now we do for the right-side:







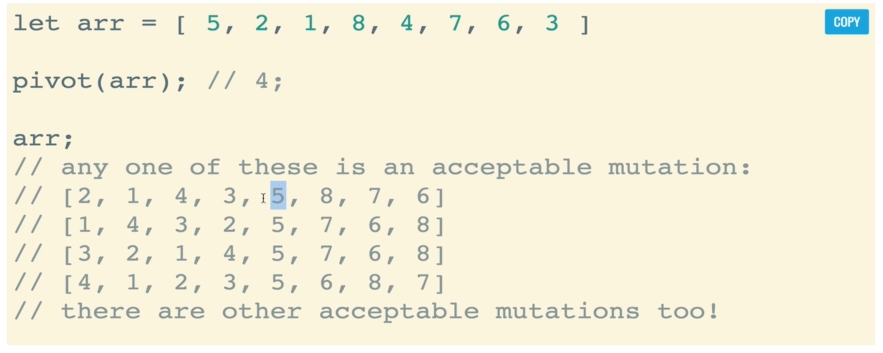
After several steps:



* The key point is, we do not swap the smaller values with pivot. We first swap the ***higher values*** with the ***smaller values*** and after find *all the smaller values* we finally swap the Pivot with the last found smaller values index.

**6.6.3 Pseudocode (part 1)** pivot()

* Pivot-Helper function:
* In order to implement Quick sort, it's useful to first implement a *function* responsible *arranging* elements in an array on either side of a pivot
* Given an array, this *helper function* should designate an *element* as the *pivot*
* It should then rearrange elements in the array so that all values less than the pivot are moved to the left of the pivot, and all values greater than the pivot are moved to the right of the pivot
* The order of elements on either side of the pivot doesn't matter!
* The helper should do this in place, that is, it should not create a new array
* When complete, the helper should return the index of the pivot
* Picking a pivot:
* The runtime of Quick Sort depends in part on how one selects the pivot
* Ideally, the pivot should be chosen so that it's roughly the median value in the data set you're sorting
* For simplicity, we'll always choose the pivot to be the first element (we'll talk about consequences of this later, in Big-O part).
* Consider the following example:
* We set 5 as the pivot.
* The main point when we call ***pivot()*** function is,
* In all mutations, **5** (**pivot**) is in the correct spot, the fixed-sorted place
* And the *index* of 5 is *always same*. The ***pivot()*** function should return this index.
* The order of the left/right moved elements doesn't matter. The values *less than pivot* are in left side and *greater than pivot* are in right side
* In the final sorted array while we doing the process over and over, the each returned index of the ***pivots*** *of each step* doesn't change.



* Pseudocode of **pivot()**: Following is the Pseudocode of a version of **pivot()**. Of course there are other versions.

|  |  |
| --- | --- |
| 1. **pivot()** will help to accept three arguments: an ***array***, a ***start index***, and an ***end index*** (these can default to 0 and the array length minus 1, respectively) 2. Grab the pivot from the start of the array (pivot can be anywhere, for simplicity we set it at start). 3. Store the current *pivot index* in a *variable* (this will *keep track* of where the pivot *should end up*) 4. Loop through the array from the start until the end 5. If the pivot is greater than the current element, *increment* the *pivot index variable* and then swap the current element (which is less than pivot) with the elementat the pivot index *(incremented pivot index variables value)*. [Actually we putting smaller values next to the pivot by swapping with higher values]. 6. After the loop is done, swap the starting element (i.e. the pivot) with the final pivot index 7. Return the pivot index |  |

**Instructor version**

// *First Version*

**function** **pivot**(arr, start=0, end=arr**.**length+1){

**function** **swap**(array, i, j) {

**var** temp=array[i];

    array[i] = array[j];

    array[j] = temp;

  }

**var** pv=arr[start];

**var** swapIdx=start;

**for**(**var** i=start+1; i **<** arr**.**length; i++){

**if**(pv **>** arr[i]){

      swapIdx++;

**swap**(arr,swapIdx,i);

      // *console.log(arr);*

    }

  }

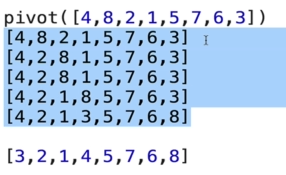
**swap**(arr,start,swapIdx);

  // *console.log(arr);*

**return** swapIdx;

}

**pivot**([4,8,2,1,5,7,6,3])



**6.6.4 Pseudocode (part 2)** quickSort()

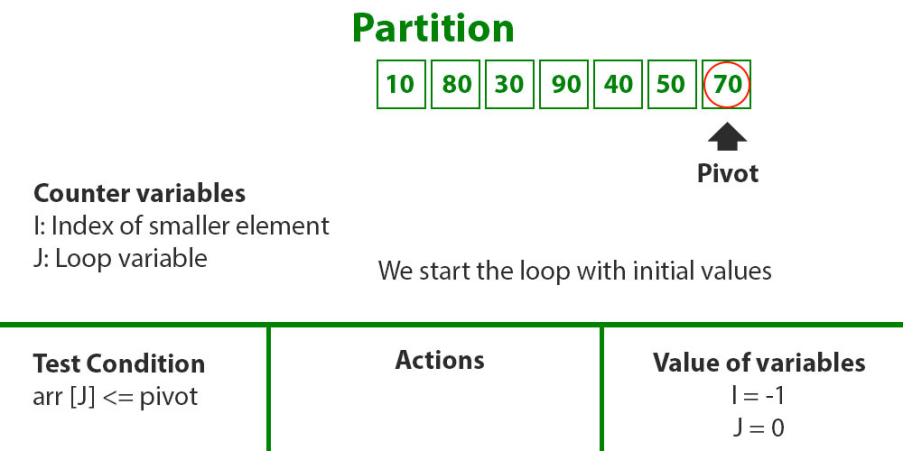
* Quicksort Pseudocode:

1. Call the pivot helper on the array
   * Call the **pivot()** helper function at the beginning. We also define some default value.
2. When the helper returns to you the updated *pivot index*, recursively call the *pivot helper* on the *subarray* to the *left of that index*, and the *subarray* to the *right of that index*.
   * After find the pivot, we do a recursive call on left side and right side of the array.
   * We do not include the pivot.
3. Your base case occurs when you consider a subarray with less than 2 elements

* We do not create any new array, we just re-arranging the values.
* So our base case needs to define on the main array (unlike merge sort), instead of checking *single valued sub-array*. So we are not going to use the *length of the array*.
* We define a base case for which the recursive call should end (e.g. ***left<right***), using the ***left*** & ***right*** ***pointer***, because these pointers move towards each other. We can also check ***left=right*** in case the sub-array is less than two elements.

|  |  |
| --- | --- |
| **function** **quickSort**(arr, left = 0, right = arr**.**length -1){  **if**(left **<** right){  **let** pivotIndex= **pivot**(arr,left,right)//*3*          //*left*  **quickSort**(arr,left,pivotIndex-1);          //*right*  **quickSort**(arr,pivotIndex+1,right);        }  **return** arr;  } |  |

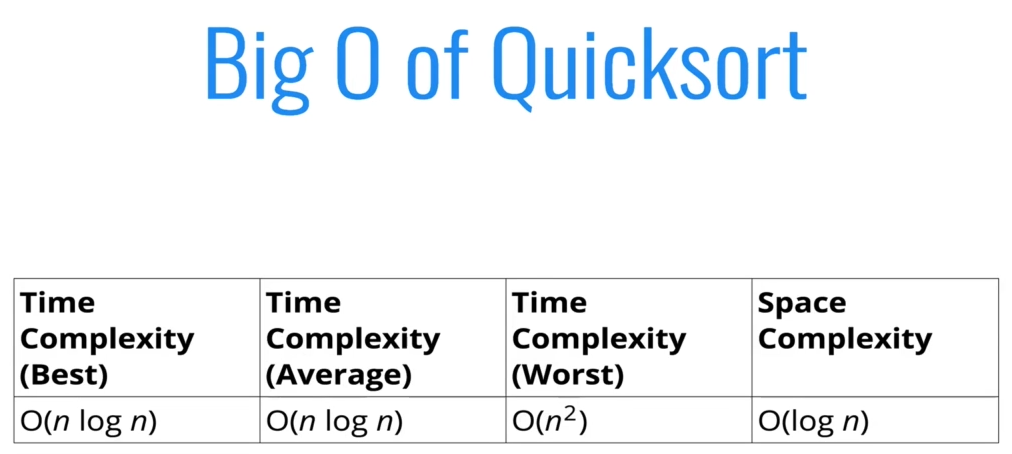
* **if**(left **<** right) defines the base case, where two pointers left and right collide.
* **quickSort**(arr,left,pivotIndex-1) makes *recursive call* on the *left sub-array*, where **pivot()** is called on the sub-array.
* **quickSort**(arr,pivotIndex+1,right) makes *recursive call* on the *right sub-array*, where **pivot()** is called on the sub-array.
* **return** arr finally returns the sorted array.
* **The call-stack:** The recursive-call first hit on the left-sides until the base case, after base case, right-side is called and stack is done.
* In following case the right-most element is set as pivot.

****

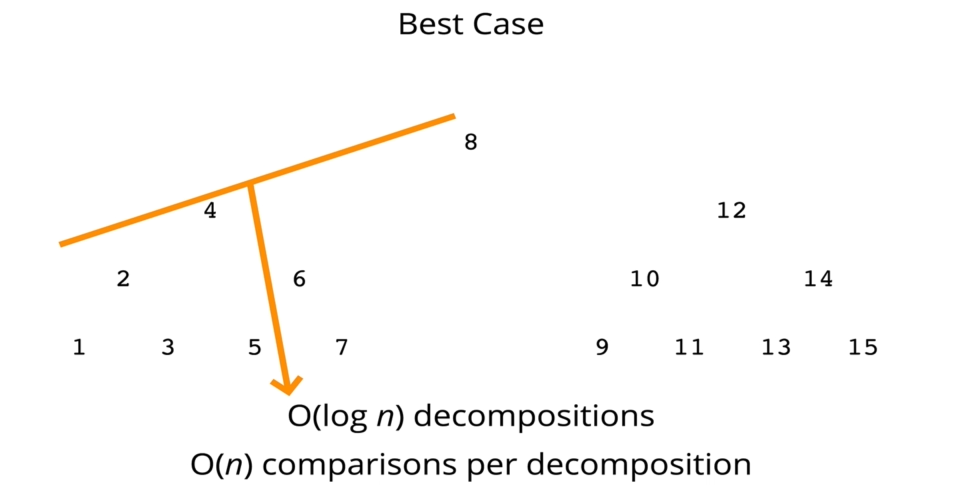
**Call stack**

****

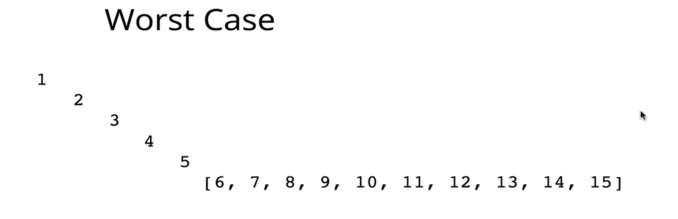
**6.6.5 Big-O for Quick sort**

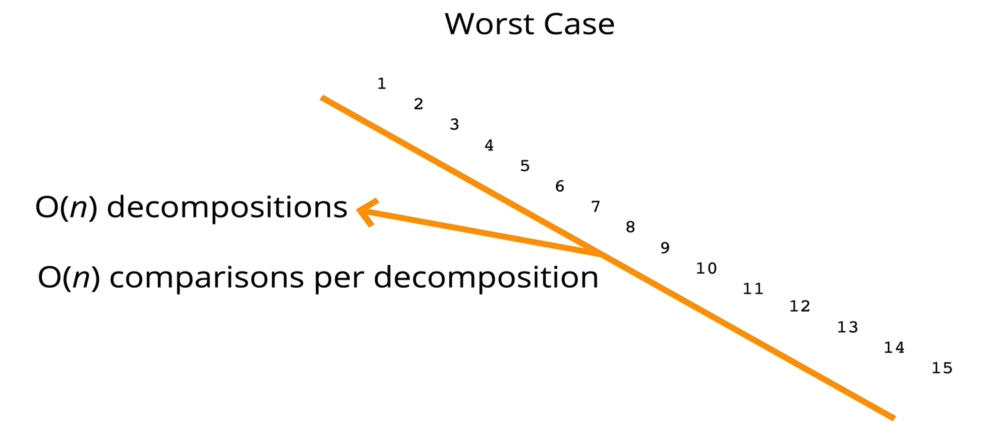


* Best Case: Makes O(log(n)) *decomposition* and **O(n)** *comparison-per-decomposition*, so it makes it **O(nlog(n))** time-complexity for average or best case.



* Worst case: Data is already sorted. Makes O(n) *decomposition* and **O(n)** *comparison-per-decomposition*, so it makes it **O(n2)** time-complexity.
* Consider the already sorted array: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15].





* The way around this worst case is:
* Pick a random pivot: But in this case we can accidently pick the min element in case of *nearly-sorted array*.
* Pick mid-element or median always: This is the best solution for *sorted* or *nearly-sorted arrays*.

**All code at once (Instructor)**

**function** **pivot**(arr, start = 0, end = arr**.**length - 1) {

**const swap** =(arr, idx1, idx2) **=>** {

[arr[idx1],arr[idx2]]=[arr[idx2],arr[idx1]];

};

  // *We are assuming the pivot is always the first element*

**let** pv=arr[start];

**let** swapIdx=start;

**for** (**let** i=start+1; i **<=** end; i++) {

**if** (pv **>** arr[i]) {

      swapIdx++;

**swap**(arr, swapIdx, i);

    }

  }

  // *Swap the pivot from the start the swapPoint*

**swap**(arr, start, swapIdx);

**return** swapIdx;

}

**function** **quickSort**(arr, left = 0, right = arr**.**length -1){

**if**(left **<** right){

**let** pivotIndex= **pivot**(arr,left,right)//*3*

        //*left*

**quickSort**(arr,left,pivotIndex-1);

        //*right*

**quickSort**(arr,pivotIndex+1,right);

      }

**return** arr;

}

**quickSort**([100,-3,2,4,6,9,1,2,5,3,23])

// *[4,6,9,1,2,5,3]*

// *[3,2,1,4,6,9,5]*

// *4*

// *3,2,1    6,9,5*

// *3      6*

// *2,1      5  9*

// *2*

// *1*