Quick Sort

Quick sort is a sorting algorithm, used in data structures for sorting arrays, queues, linked lists and other linear data structures. Quick sort works on the principle of Divide and Conquer. Quick sort is also known as partition-exchange sort.

Time Complexity	O(n log n)
Best Case	O(n log n)
Worst Case	O(n2)
Space Complexity	O(1)
Auxiliary Space Complexity	O(log n)
In Place	Yes
Stable	No

Quick Sort in Java

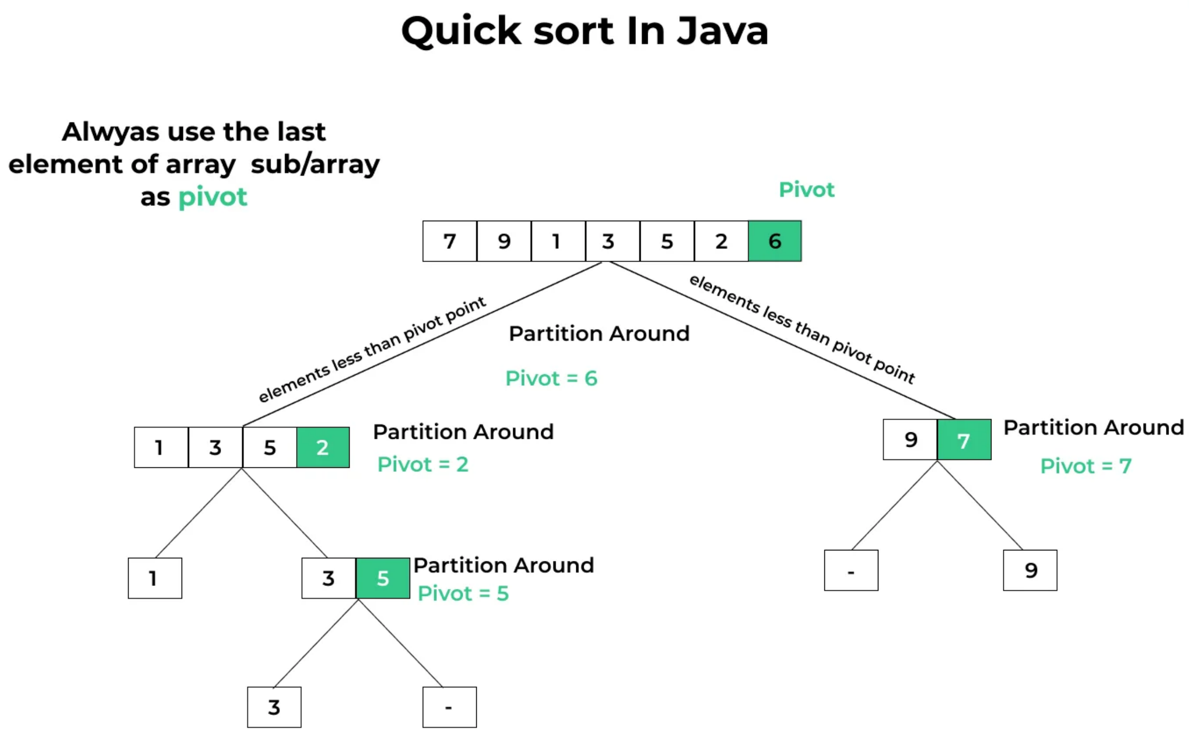
- Choose the pivot element.
- Move the elements smaller to pivot in the left partition
- Move the elements greater than pivot to the right partition
- The partition index is discovered at the end
-

New pivot is then chosen in each of the partition and the above steps are repeated until partitions of one item each is reached.

Generally, pivot can be chosen as any of the following –

- Last element
- First element
- Random element
- Median element

We will choose the last element as a pivot



Steps to implement Quick sort:

1) Choose an element, called pivot, from the list. Generally pivot can be the last index element.

2)Reorder the list so that all elements with values less than the pivot come before the pivot, while all elements with values greater than the pivot come after it . After this partitioning, the pivot is in its final position. This is called the partition operation.

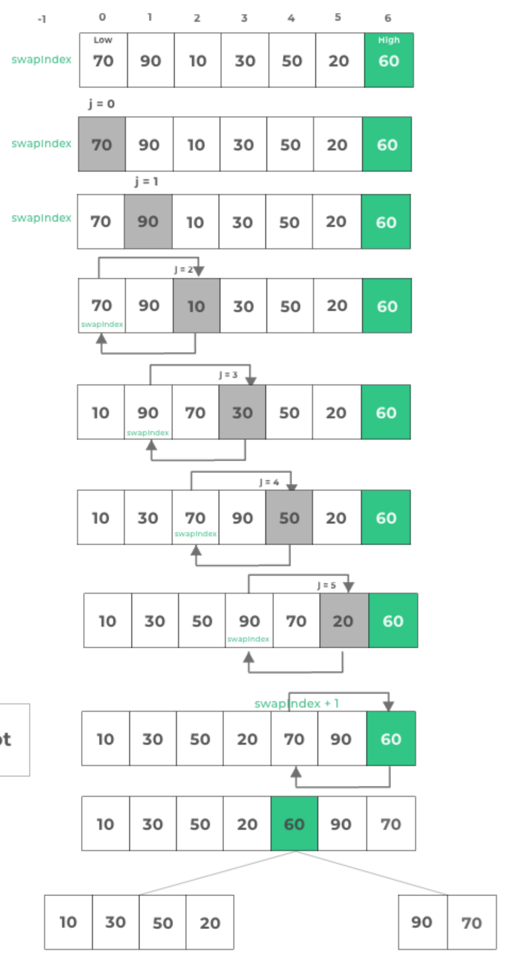
3)Recursively apply the above steps to the sub-list of elements with smaller values and separately the sub-list of elements with greater values.

Quick Sort in Java

pivot = High

pivot = High

swapIndex = low - 1



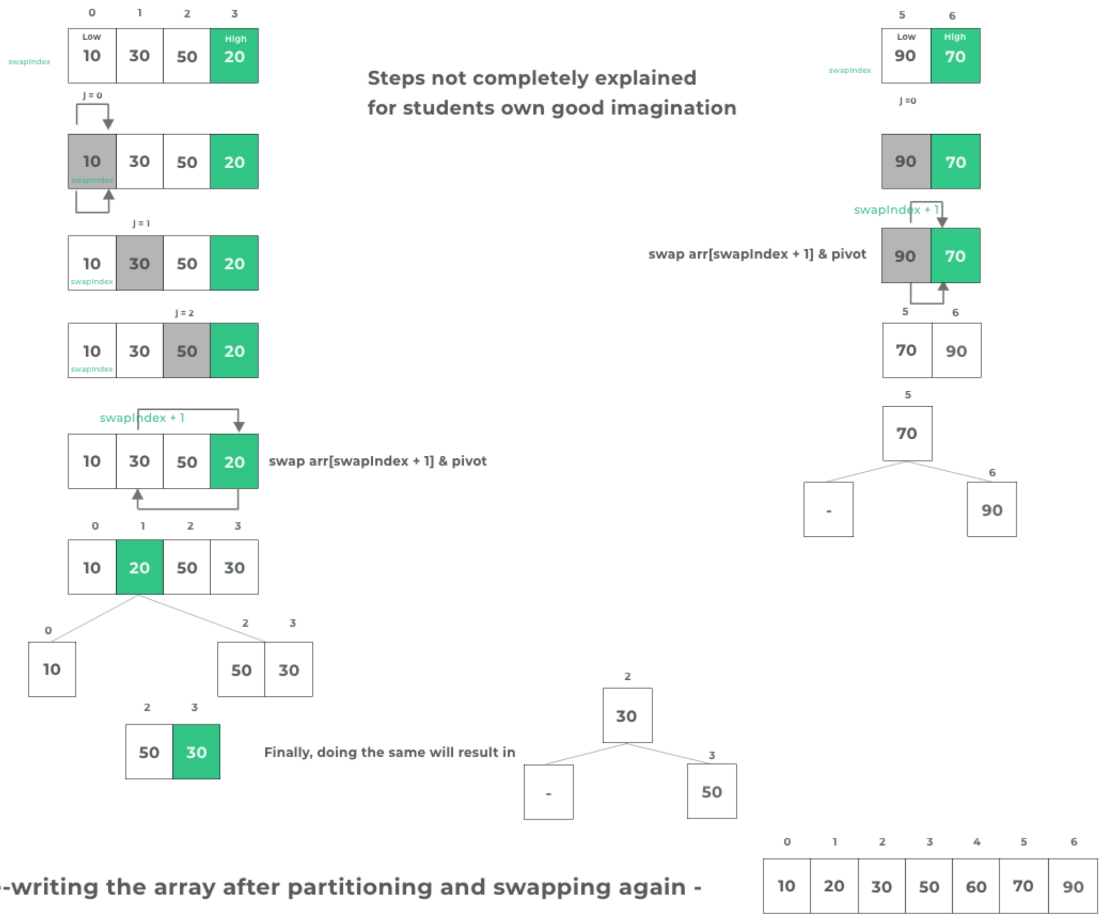
arr[j] < pivot	Action	
No	None	swapIndex = -1
No	None	swapIndex = -1
Yes	swapIndex++ Swap 10 & 70	swapIndex = 0
Yes	swapIndex++ Swap 30 & 90	swapIndex = 1
Yes	swapIndex++ Swap 50 & 70	swapIndex = 2
Yes	swapIndex++ Swap 20 & 90	swapIndex = 3

swap arr[swapIndex + 1] & pivot

Left Partition

Right Partition

Quick Sort in Java



swapIndex
Low
High
j = 0
j = 1
j = 2
swapIndex + 1
swap arr[swapIndex + 1] & pivot
Steps not completely explained
for students own good imagination
Finally, doing the same will result in
Re-writing the array after partitioning and swapping again -

```
package Sort;

public class Quick {
  // this function display the array
  public static void printArray(int[] arr, int size) {
    for (int i = 0; i < size; i++) {
      System.out.print(arr[i] + " ");
    }
    System.out.println();
  }

  // main function of the program
  public static void main(String[] args) {
    int[] a = { 12, 11, 13, 5, 6, 7 };

    int size = a.length;
    System.out.println("Array Before Sort:");
    printArray(a, size);

    quickSort(a, 0, size - 1);

    System.out.println("Array After Sort:");
    printArray(a, size);
  }

  // A utility function to swap two elements
  static void swap(int[] arr, int i, int j) {
    int temp = arr[i];
    arr[i] = arr[j];
    arr[j] = temp;
  }

  // Recursive function to apply quickSort
  static void quickSort(int[] arr, int low, int high) {
    if (low < high) {
      /*
       * indexPI is partitioning index, partition() function will return index of
       * partition
       */
      int indexPI = partition(arr, low, high);

      quickSort(arr, low, indexPI - 1); // left partition
      quickSort(arr, indexPI + 1, high); // right partition
    }
  }

  /*
   * Partition function to do Partition elements on the left side of pivot
   * elements would be smaller than pivot elements on the right side of pivot
   * would be greater than the pivot
   */
  static int partition(int[] arr, int low, int high) {
    // Pivot element selected as right most element in array each time.
    int pivot = arr[high];
    int swapIndex = (low - 1); // swapping index.

    for (int j = low; j <= high - 1; j++) {
      // Check if current element is smaller than pivot element.
      if (arr[j] < pivot) {
        swapIndex++; // increment swapping index.
        swap(arr, swapIndex, j);
      }
    }
    // swap swapindex+ 1 and pivot index
    // we assigned pivot = arr[high] is pivot index is high
    swap(arr, swapIndex + 1, high);

    return (swapIndex + 1);
  }
}
```

- This algorithm is quite efficient for large-sized data sets as its average and worst-case complexity are of $\mathrm{O}(n^2)$, where n is the number of items.
- Quick sort is more advantageous than merge sort, as in merge sort we need a temporary array to merge the sorted arrays, and hence it is not fruitful than quick

sort.

Disadvantages

- Needs additional spaces for temporary arrays, thus space complexity is O(log n)
- If the first element is chosen as a pivot it causes worst-case complexity of $O(n^2)$