**Implement Various Types of Partitions in Quick Sort in Java**

* Difficulty Level : [Expert](https://www.geeksforgeeks.org/expert/)
* Last Updated : 07 Aug, 2021

[Quicksort](https://www.geeksforgeeks.org/quick-sort/) is a [Divide and Conquer](https://www.geeksforgeeks.org/divide-and-conquer-algorithm-introduction/) Algorithm that is used for sorting the elements. In this algorithm, we choose a pivot and partitions the given array according to the pivot. Quicksort algorithm is a mostly used algorithm because this algorithm is cache-friendly and performs in-place sorting of the elements means no extra space requires for sorting the elements.

**Note:**

*Quicksort algorithm is generally unstable algorithm because quick sort cannot be able to maintain the relative   
order of the elements.*

**Three partitions are possible for the Quicksort algorithm:**

1. **Naive partition:** In this partition helps to maintain the relative order of the elements but this partition takes O(n) extra space.
2. **Lomuto partition:**In this partition, The last element chooses as a pivot in this partition. The pivot acquires its required position after partition but more comparison takes place in this partition.
3. **Hoare’s partition:** In this partition, The first element chooses as a pivot in this partition. The pivot displaces its required position after partition but less comparison takes place as compared to the Lomuto partition.

**1.  Naive partition**

**Algorithm:**

**Naivepartition(arr[],l,r)**

1. Make a Temporary array temp[r-l+1] length

2. Choose last element as a pivot element

3. Run two loops:

-> Store all the elements in the temp array that are less than pivot element

-> Store the pivot element

-> Store all the elements in the temp array that are greater than pivot element.

4.Update all the elements of arr[] with the temp[] array

**QuickSort(arr[], l, r)**

If r > l

1. Find the partition point of the array

m = Naivepartition(a,l,r)

2. Call Quicksort for less than partition point

Call Quicksort(arr, l, m-1)

3. Call Quicksort for greater than the partition point

Call Quicksort(arr, m+1, r)

* Java

|  |
| --- |
| // Java program to demonstrate the naive partition  // in quick sort    **import** java.io.\*;  **import** java.util.\*;  **public** **class** GFG {  **static** **int** partition(**int** a[], **int** start, **int** high)      {          // Creating temporary  **int** temp[] = **new** **int**[(high - start) + 1];            // Choosing a pivot  **int** pivot = a[high];  **int** index = 0;            // smaller number  **for** (**int** i = start; i <= high; ++i) {  **if** (a[i] < pivot)              {                  temp[index++] = a[i];              }          }            // pivot position  **int** position = index;            // Placing the pivot to its original position          temp[index++] = pivot;    **for** (**int** i = start; i <= high; ++i)          {  **if** (a[i] > pivot)              {                  temp[index++] = a[i];              }          }            // Change the original array  **for** (**int** i = start; i <= high; ++i) {              a[i] = temp[i - start];          }            // return the position of the pivot  **return** position;      }    **static** **void** quicksort(**int** numbers[], **int** start, **int** end)      {  **if** (start < end) {  **int** point = partition(numbers, start, end);                quicksort(numbers, start, point - 1);              quicksort(numbers, point + 1, end);          }      }        // Function to print the array  **static** **void** print(**int** numbers[])      {  **for** (**int** a : numbers)          {              System.out.print(a + " ");          }      }    **public** **static** **void** main(String[] args)      {  **int** numbers[] = { 3, 2, 1, 78, 9798, 97 };            // rearrange using naive partition          quicksort(numbers, 0, numbers.length - 1);            print(numbers);      }  } |

**Output**

1 2 3 78 97 9798

**2.  Lomuto partition**

* **Lomuto’s Partition Algorithm (**[**unstable**](https://www.geeksforgeeks.org/stability-in-sorting-algorithms/)**algorithm)**

**Lomutopartition(arr[], lo, hi)**

pivot = arr[hi]

i = lo // place for swapping

for j := lo to hi – 1 do

if arr[j] <= pivot then

swap arr[i] with arr[j]

i = i + 1

swap arr[i] with arr[hi]

return i

**QuickSort(arr[], l, r)**

If r > l

1. Find the partition point of the array

m =Lomutopartition(a,l,r)

2. Call Quicksort for less than partition point

Call Quicksort(arr, l, m-1)

3. Call Quicksort for greater than the partition point

Call Quicksort(arr, m+1, r)

* Java

|  |
| --- |
| // Java program to demonstrate the Lomuto partition  // in quick sort    **import** java.util.\*;  **public** **class** GFG {    **static** **int** sort(**int** numbers[], **int** start, **int** last)      {  **int** pivot = numbers[last];  **int** index = start - 1;  **int** temp = 0;    **for** (**int** i = start; i < last; ++i)          {  **if** (numbers[i] < pivot) {                  ++index;                    // swap the position                  temp = numbers[index];                  numbers[index] = numbers[i];                  numbers[i] = temp;              }          }    **int** pivotposition = ++index;            temp = numbers[index];          numbers[index] = pivot;          numbers[last] = temp;    **return** pivotposition;      }    **static** **void** quicksort(**int** numbers[], **int** start, **int** end)      {  **if** (start < end)          {  **int** pivot\_position = sort(numbers, start, end);              quicksort(numbers, start, pivot\_position - 1);              quicksort(numbers, pivot\_position + 1, end);          }      }    **static** **void** print(**int** numbers[])      {  **for** (**int** a : numbers) {              System.out.print(a + " ");          }      }    **public** **static** **void** main(String[] args)      {  **int** numbers[] = { 4, 5, 1, 2, 4, 5, 6 };          quicksort(numbers, 0, numbers.length - 1);          print(numbers);      }  } |

**Output**

1 2 4 4 5 5 6

**3.  Hoare’s Partition**

[Hoare’s Partition Scheme](https://en.wikipedia.org/wiki/Quicksort#Hoare_partition_scheme) works by initializing two indexes that start at two ends, the two indexes move toward each other until an inversion is (A smaller value on the left side and a greater value on the right side) found. When an inversion is found, two values are swapped and the process is repeated.

**Algorithm:**

**Hoarepartition(arr[], lo, hi)**

pivot = arr[lo]

i = lo - 1 // Initialize left index

j = hi + 1 // Initialize right index

// Find a value in left side greater

// than pivot

do

i = i + 1

while arr[i] < pivot

// Find a value in right side smaller

// than pivot

do

j--;

while (arr[j] > pivot);

if i >= j then

return j

swap arr[i] with arr[j]

**QuickSort(arr[], l, r)**

If r > l

1. Find the partition point of the array

m =Hoarepartition(a,l,r)

2. Call Quicksort for less than partition point

Call Quicksort(arr, l, m)

3. Call Quicksort for greater than the partition point

Call Quicksort(arr, m+1, r)

* Java

|  |
| --- |
| // Java implementation of QuickSort  // using Hoare's partition scheme    **import** java.io.\*;    **class** GFG {        // This function takes first element as pivot, and      // places all the elements smaller than the pivot on the      // left side and all the elements greater than the pivot      // on the right side. It returns the index of the last      // element on the smaller side  **static** **int** partition(**int**[] arr, **int** low, **int** high)      {  **int** pivot = arr[low];  **int** i = low - 1, j = high + 1;    **while** (**true**)          {              // Find leftmost element greater              // than or equal to pivot  **do** {                  i++;              } **while** (arr[i] < pivot);                // Find rightmost element smaller              // than or equal to pivot  **do** {                  j--;              } **while** (arr[j] > pivot);                // If two pointers met.  **if** (i >= j)  **return** j;                // swap(arr[i], arr[j]);  **int** temp = arr[i];              arr[i] = arr[j];              arr[j] = temp;            }      }        // The main function that      // implements QuickSort      // arr[] --> Array to be sorted,      // low --> Starting index,      // high --> Ending index  **static** **void** quickSort(**int**[] arr, **int** low, **int** high)      {  **if** (low < high) {                // pi is partitioning index,              // arr[p] is now at right place  **int** pi = partition(arr, low, high);                // Separately sort elements before              // partition and after partition              quickSort(arr, low, pi);              quickSort(arr, pi + 1, high);          }      }        // Function to print an array  **static** **void** printArray(**int**[] arr, **int** n)      {  **for** (**int** i = 0; i < n; ++i)              System.out.print(" " + arr[i]);            System.out.println();      }        // Driver Code  **static** **public** **void** main(String[] args)      {  **int**[] arr = { 10, 17, 18, 9, 11, 15 };  **int** n = arr.length;          quickSort(arr, 0, n - 1);            printArray(arr, n);      }  } |

**Output**

9 10 11 15 17 18