Hoare's vs Lomuto partition scheme in QuickSort - GeeksforGeeks

Source: https://www.geeksforgeeks.org/hoares-vs-lomuto-partition-scheme-quicksort/

Courses Tutorials Practice Jobs DSA Tutorial Interview Questions Quizzes Must Do Advanced DSA System Design Aptitude Puzzles Interview Corner DSA Python Technical Scripter 2026 Explore DSA Fundamentals Logic Building Problems Analysis of Algorithms Data Structures Array Data Structure String in Data Structure Hashing in Data Structure Linked List Data Structure Stack Data Structure Queue Data Structure Tree Data Structure Graph Data Structure Trie Data Structure Algorithms Searching Algorithms Sorting Algorithms Introduction to Recursion Greedy Algorithms Tutorial Graph Algorithms Dynamic Programming or DP Bitwise Algorithms Advanced Segment Tree Binary Indexed Tree or Fenwick Tree Square Root (Sqrt) Decomposition Algorithm Binary Lifting Geometry Interview Preparation Interview Corner GfG160 Practice Problem GeeksforGeeks Practice - Leading Online Coding Platform Problem of The Day - Develop the Habit of Coding DSA Course 90% Refund Hoare's vs Lomuto partition scheme in QuickSort Last Updated : 23 Jul, 2025 We have discussed the implementation of QuickSort using Lomuto partition scheme . Lomuto's partition scheme is easy to implement as compared to Hoare scheme. This has inferior performance to Hoare's QuickSort. Lomuto's Partition Scheme: This algorithm works by assuming the pivot element as the last element. If any other element is given as a pivot element then swap it first with the last element. Now initialize two variables i as low and j also low, iterate over the array and increment i when arr[j] <= pivot and swap arr[i] with arr[j] otherwise increment only j. After coming out from the loop swap arr[i+1] with arr[hi]. This i stores the pivot element. partition(arr[], lo, hi) pivot = arr[hi] i = lo-1 // place for swapping for j := lo to hi – 1 do if arr[j] <= pivot then i = i + 1 swap arr[i] with arr[j] swap arr[i+1] with arr[hi] return i+1 Refer QuickSort for details of this partitioning scheme. Below are implementations of this approach:- C++ #include <bits/stdc++.h> using namespace std ; /* This function takes the last element as pivot, places the pivot element at its correct position in sorted array, and places all smaller (smaller than pivot) to the left of the pivot and all greater elements to the right of the pivot */ int partition (vector < int >& arr , int low , int high) { int pivot = arr [high]; // pivot int i = (low - 1); // Index of smaller element for (int j = low ; j <= high - 1 ; j ++) { // If current element is smaller than or // equal to pivot if (arr [j] <= pivot) { i ++ ; // increment index of smaller element swap (arr [i], arr [j]); } } swap (arr [i + 1], arr [high]); return (i + 1); } /* The main function that implements QuickSort arr[] --> Array to be sorted, low --> Starting index, high --> Ending index */ void quickSort (vector < 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 - 1); quickSort (arr , pi + 1 , high); } } /* Function to print an array */ void printArray (const vector < int >& arr) { for (int i : arr) cout << i << " " ; cout << endl ; } // Driver program to test above functions int main () { vector < int > arr = { 10 , 7 , 8 , 9 , 1 , 5 }; int n = arr . size (); quickSort (arr , 0 , n - 1); cout << "Sorted array: \n " ; printArray (arr); return 0 ; } C #include <stdio.h> #include <stdlib.h> // Function to swap two elements void swap (int * a , int * b) { int t = * a ; * a = * b ; * b = t ; } /* This function takes the last element as pivot, places the pivot element at its correct position in sorted array, and places all smaller (smaller than pivot) to the left of the pivot and all greater elements to the right of the pivot */ int partition (int arr [], int low , int high) { int pivot = arr [high]; // pivot int i = (low - 1); // Index of smaller element for (int j = low ; j <= high - 1 ; j ++) { // If current element is smaller than or // equal to pivot if (arr [j] <= pivot) { i ++ ; // increment index of smaller element swap (& arr [i], & arr [j]); } } swap (& arr [i + 1], & arr [high]); return (i + 1); } /* The main function that implements QuickSort arr[] --> Array to be sorted, low --> Starting index, high --> Ending index */ 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 - 1); quickSort (arr , pi + 1 , high); } } /* Function to print an array */ void printArray (int arr [], int size) { for (int i = 0 ; i < size ; i ++) printf ("%d " , arr [i]); printf (" \n "); } // Driver program to test above functions int main () { int arr [] = { 10 , 7 , 8 , 9 , 1 , 5 }; int n = sizeof (arr) / sizeof (arr [0]); quickSort (arr , 0 , n - 1); printf ("Sorted array: \n "); printArray (arr , n); return 0 ; } Java // Java implementation QuickSort // using Lomuto's partition Scheme import java.io.* ; class GFG { static void Swap (int [] array , int position1 , int position2) { // Swaps elements in

an array // Copy the first position's element int temp = array [position1] ; // Assign to the second element array [position1] = array [position2] ; // Assign to the first element array [position2] = temp ; } /* This function takes last element as pivot, places the pivot element at its correct position in sorted array, and places all smaller (smaller than pivot) to left of pivot and all greater elements to right of pivot */ static int partition (int [] arr , int low , int high) { int pivot = arr [high] ; // Index of smaller element int i = (low - 1); for (int j = low ; j <= high - 1 ; j ++) { // If current element is smaller // than or equal to pivot if (arr [j] <= pivot) { i ++ ; // increment index of // smaller element Swap (arr , i , j); } } Swap (arr , i + 1 , high); return (i + 1); } /* 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 - 1); quickSort (arr , pi + 1 , high); } } /* Function to print an array */ static void printArray (int [] arr , int size) { int i ; for (i = 0 ; i < size ; i ++) System . out . print (" " + arr [i]); System . out . println (); } // Driver Code static public void main (String [] args) { int [] arr = { 10 , 7 , 8 , 9 , 1 , 5 }; int n = arr . length ; quickSort (arr , 0 , n - 1); System . out . println ("Sorted array: "); printArray (arr , n); } } // This code is contributed by vt_m. Python ''' Python3 implementation QuickSort using Lomuto's partition Scheme.''' ''' This function takes last element as pivot, places the pivot element at its correct position in sorted array, and places all smaller (smaller than pivot) to left of pivot and all greater elements to right of pivot ''' def partition (arr , low , high): # pivot pivot = arr [high] # Index of smaller element i = (low - 1) for j in range (low , high): # If current element is smaller than or # equal to pivot if (arr [j] <= pivot): # increment index of smaller element i += 1 arr [i], arr [j] = arr [j], arr [i] arr [i + 1], arr [high] = arr [high], arr [i + 1] return (i + 1) ''' The main function that implements QuickSort arr --> Array to be sorted, low --> Starting index, high --> Ending index ''' def quickSort (arr , low , high): if (low < high): ''' pi is partitioning index, arr[p] is now at right place ''' pi = partition (arr , low , high) # Separately sort elements before # partition and after partition quickSort (arr , low , pi - 1) quickSort (arr , pi + 1 , high) ''' Function to print an array ''' def printArray (arr , size): for i in range (size): print (arr [i], end = " ") print () # Driver code arr = [10 , 7 , 8 , 9 , 1 , 5] n = len (arr) quickSort (arr , 0 , n - 1) print ("Sorted array:") printArray (arr , n) # This code is contributed by SHUBHAMSINGH10 C# // C# implementation QuickSort // using Lomuto's partition Scheme using System ; class GFG { static void Swap (int [] array , int position1 , int position2) { // Swaps elements in an array // Copy the first position's element int temp = array [position1]; // Assign to the second element array [position1] = array [position2]; // Assign to the first element array [position2] = temp ; } /* This function takes last element as pivot, places the pivot element at its correct position in sorted array, and places all smaller (smaller than pivot) to left of pivot and all greater elements to right of pivot */ static int partition (int [] arr , int low , int high) { int pivot = arr [high]; // Index of smaller element int i = (low - 1); for (int j = low ; j <= high - 1 ; j ++) { // If current element is smaller // than or equal to pivot if (arr [j] <= pivot) { i ++ ; // increment index of // smaller element Swap (arr , i , j); } } Swap (arr , i + 1 , high); return (i + 1); } /* 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 - 1); quickSort (arr , pi + 1 , high); } } /* Function to print an array */ static void printArray (int [] arr , int size) { int i ; for (i = 0 ; i < size ; i ++) Console . Write (" " + arr [i]); Console . WriteLine (); } // Driver Code static public void Main () { int [] arr = { 10 , 7 , 8 , 9 , 1 , 5 }; int n = arr . Length ; quickSort (arr , 0 , n - 1); Console . WriteLine ("Sorted array: "); printArray (arr , n); } } // This code is contributed by vt_m. JavaScript /* This function takes the last element as pivot, places the pivot element at its correct position in sorted array, and places all smaller (smaller than pivot) to the left of the pivot and all greater elements to the right of the pivot */ function partition (arr , low , high) { let pivot = arr [high]; // pivot let i = low - 1 ; // Index of smaller element for (let j = low ; j <= high - 1 ; j ++) { // If current element is smaller than or // equal to pivot if (arr [j] <= pivot) { i ++ ; // increment index of smaller element // Swap arr[i] and arr[j] [arr [i], arr [j]] = [arr [j], arr [i]]; } } // Swap arr[i + 1] and arr[high] [arr [i + 1], arr [high]] = [arr [high], arr [i + 1]]; return i + 1 ; } /* The main function that implements QuickSort arr[] --> Array to be sorted, low --> Starting index, high --> Ending index */ function quickSort (arr , low , high) { if (low < high) { /* pi is partitioning index, arr[pi] is now at right place */ let pi = partition (arr , low , high); // Separately sort elements before // partition and after partition quickSort (arr , low , pi - 1); quickSort (arr , pi + 1 , high); } } // Driver code to test above functions let arr = [10 , 7 , 8 , 9 , 1 , 5]; let n = arr . length ; quickSort (arr , 0 , n - 1); console . log ("Sorted array:"); console . log (arr . join (' ')); Output Sorted array: 1 5 7 8 9 10 Time Complexity: O(N

2) Auxiliary Space: O(1) Hoare's Partition Scheme: Hoare's 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 greater value on the right side) found. When an inversion is found, two values are swapped and the process is repeated. Algorithm: partition(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] Below are implementations of this approach:- C++ #include <bits/stdc++.h> using namespace std ; /* This function takes the 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 */ int partition (vector < 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]); } } /* The main function that implements QuickSort arr[] --> Array to be sorted, low --> Starting index, high --> Ending index */ void quickSort (vector < int >& arr , int low , int high) { if (low < high) { /* pi is partitioning index, arr[pi] 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 */ void printArray (const vector < int >& arr) { for (int i : arr) cout << i << " " ; cout << endl ; } // Driver Code int main () { vector < int > arr = { 10 , 7 , 8 , 9 , 1 , 5 }; quickSort (arr , 0 , arr . size () - 1); cout << "Sorted array: \n " ; printArray (arr); return 0 ; } C #include <stdio.h> /* This function takes the 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 */ int partition (int arr [], int low , int high) { int pivot = arr [low]; int i = low - 1 , j = high + 1 ; while (1) { // 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] and 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 */ void quickSort (int arr [], int low , int high) { if (low < high) { /* pi is partitioning index, arr[pi] 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 */ void printArray (int arr [], int size) { for (int i = 0 ; i < size ; i ++) printf ("%d " , arr [i]); printf (" \n "); } // Driver code to test above functions int main () { int arr [] = { 10 , 7 , 8 , 9 , 1 , 5 }; int n = sizeof (arr) / sizeof (arr [0]); quickSort (arr , 0 , n - 1); printf ("Sorted array: \n "); printArray (arr , n); return 0 ; } 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 ; int temp = arr [i] ; arr [i] = arr [j] ; arr [j] = temp ; // swap(arr[i], arr[j]); } } /* 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 , 7 , 8 , 9 , 1 , 5 }; int n = arr . length ; quickSort (arr , 0 , n - 1); System . out . println ("Sorted array: "); printArray (arr , n); } } // This code is contributed by vt_m. Python ''' Python implementation of QuickSort using Hoare's partition scheme. ''' ''' 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 ''' def partition (arr , low , high): pivot = arr [low] i = low - 1 j = high + 1 while (True): # Find leftmost element greater than # or equal to pivot i += 1 while (arr [i] < pivot): i += 1 # Find rightmost element smaller than # or equal to pivot j -= 1 while (arr [j] > pivot): j -= 1 # If two pointers met. if (i >= j): return j arr [i], arr [j] = arr [j], arr [i] ''' The main function that implements QuickSort arr --> Array to be sorted, low --> Starting index, high --> Ending index ''' def quickSort (arr , low , high): ''' pi is partitioning index, arr[p] is now at right place ''' if (low < high): 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 ''' def printArray (arr , n): for i in range (n): print (arr [i], end = " ") print () # Driver code arr = [10 , 7 , 8 , 9 , 1 , 5] n = len (arr) quickSort (arr , 0 , n - 1) print ("Sorted array:") printArray (arr , n) # This code is contributed by shubhamsingh10 C# // C# implementation of QuickSort // using Hoare's partition scheme using System ; 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 ; int temp = arr [i]; arr [i] = arr [j]; arr [j] = temp ; // swap(arr[i], arr[j]); } } /* 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 ++) Console . Write (" " + arr [i]); Console . WriteLine (); } // Driver Code static public void Main () { int [] arr = { 10 , 7 , 8 , 9 , 1 , 5 }; int n = arr . Length ; quickSort (arr , 0 , n - 1); Console . WriteLine ("Sorted array: "); printArray (arr , n); } } // This code is contributed by vt_m. JavaScript /* This function takes the 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 */ function partition (arr , low , high) { let pivot = arr [low]; let 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] and arr[j] [arr [i], arr [j]] = [arr [j], arr [i]]; } } /* The main function that implements QuickSort arr[] --> Array to be sorted, low --> Starting index, high --> Ending index */ function quickSort (arr , low , high) { if (low < high) { /* pi is partitioning index, arr[pi] is now at right place */ let pi = partition (arr , low , high); // Separately sort elements before // partition and after partition quickSort (arr , low , pi); quickSort (arr , pi + 1 , high); } } // Driver code to test above functions let arr = [10 , 7 , 8 , 9 , 1 , 5]; let n = arr . length ; quickSort (arr , 0 , n - 1); console . log ("Sorted array:"); console . log (arr . join (' ')); Output Sorted array: 1 5 7 8 9 10 Time Complexity: O(N) Auxiliary Space: O(1) Note : If we change Hoare's partition to pick the last element as pivot, then the Hoare's partition may cause QuickSort to go into an infinite recursion. For example, {10, 5, 6, 20} and pivot is arr[high], then returned index will always be high and call to same QuickSort will be made. To handle a random pivot, we can always swap that random element with the first element and simply follow the above algorithm. Comparison: Hoare's scheme is more efficient than Lomuto's partition scheme because it does three times fewer swaps on average, and it creates efficient partitions even when all values are equal. Like Lomuto's partition scheme, Hoare partitioning also causes Quick sort to degrade to O(n^2) when the input array is already sorted, it also doesn't produce a stable sort. Note that in this scheme, the pivot's final location is not necessarily at the index that was returned, and the next two segments that the main algorithm recurs on are (lo..p) and (p+1..hi) as opposed to (lo..p-1) and (p+1..hi) as in Lomuto's scheme. Both Hoare's Partition, as well as Lomuto's partition, are unstable. Hoare partition algorithm Lomuto partition algorithm Generally, the first item or the element is assumed to be the initial pivot element. Some choose the middle element and even the last element. Generally, a random element of the array is located and picked and then exchanged with the first or the last element to give initial pivot values. In the aforementioned algorithm, the last element of the list is considered as the initial pivot element. It is a linear algorithm. It is also a linear algorithm. It is relatively faster. It is slower. It is slightly difficult to understand and to implement. It is easy to understand and easy to implement. It doesn't fix the pivot element in the correct position. It fixes the pivot element in the correct position. Source : https://en.wikipedia.org/wiki/Quicksort#Hoare_partition_scheme Comment Article Tags: Article Tags: Sorting DSA Quick Sort