Quick Sort - GeeksforGeeks

Source: https://www.geeksforgeeks.org/quick-sort/

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 Quick Sort Last Updated : 8 Dec, 2025 QuickSort is a sorting algorithm based on the Divide and Conquer that picks an element as a pivot and partitions the given array around the picked pivot by placing the pivot in its correct position in the sorted array. . There are mainly three steps in the algorithm: Choose a Pivot: Select an element from the array as the pivot. The choice of pivot can vary (e.g., first element, last element, random element, or median). Partition the Array: Re arrange the array around the pivot. After partitioning, all elements smaller than the pivot will be on its left, and all elements greater than the pivot will be on its right. Recursively Call: Recursively apply the same process to the two partitioned sub-arrays. Base Case: The recursion stops when there is only one element left in the sub-array, as a single element is already sorted. Choice of Pivot There are many different choices for picking pivots. Always pick the first (or last) element as a pivot . The below implementation picks the last element as pivot. The problem with this approach is it ends up in the worst case when array is already sorted. Pick a random element as a pivot . This is a preferred approach because it does not have a pattern for which the worst case happens. Pick the median element is pivot. This is an ideal approach in terms of time complexity as we can find median in linear time and the partition function will always divide the input array into two halves. But it takes more time on average as median finding has high constants. Partition Algorithm The key process in quickSort is a partition(). There are three common algorithms to partition. All these algorithms have O(n) time complexity. Naive Partition : Here we create copy of the array. First put all smaller elements and then all greater. Finally we copy the temporary array back to original array. This requires O(n) extra space. Lomuto Partition : We have used this partition in this article. This is a simple algorithm, we keep track of index of smaller elements and keep swapping. We have used it here in this article because of its simplicity. Hoare's Partition : This is the fastest of all. Here we traverse array from both sides and keep swapping greater element on left with smaller on right while the array is not partitioned. Please refer Hoare’s vs Lomuto for details. Working of Lomuto Partition Algorithm with Illustration The logic is simple, we start from the leftmost element and keep track of the index of smaller (or equal) elements as i . While traversing, if we find a smaller element, we swap the current element with arr[i] . Otherwise, we ignore the current element. Try it on GfG Practice Let us understand the working of partition algorithm with the help of the following example: Illustration of QuickSort Algorithm In the previous step, we looked at how the partitioning process rearranges the array based on the chosen pivot . Next, we apply the same method recursively to the smaller sub-arrays on the left and right of the pivot. Each time, we select new pivots and partition the arrays again. This process continues until only one element is left, which is always sorted. Once every element is in its correct position, the entire array is sorted. Below image illustrates, how the recursive method calls for the smaller sub-arrays on the left and right of the pivot : C++ #include <iostream> #include <vector> using namespace std ; int partition (vector < int >& arr , int low , int high) { // choose the pivot int pivot = arr [high]; // undex of smaller element and indicates // the right position of pivot found so far int i = low - 1 ; // Traverse arr[low..high] and move all smaller // elements on left side. Elements from low to // i are smaller after every iteration for (int j = low ; j <= high - 1 ; j ++) { if (arr [j] < pivot) { i ++ ; swap (arr [i], arr [j]); } } // move pivot after smaller elements and // return its position swap (arr [i + 1], arr [high]); return i + 1 ; } // the QuickSort function implementation void quickSort (vector < int >& arr , int low , int high) { if (low < high) { // pi is the partition return index of pivot int pi = partition (arr , low , high); // recursion calls for smaller elements // and greater or equals elements quickSort (arr , low , pi - 1);

quickSort (arr , pi + 1 , high); } } int main () { vector < int > arr = { 10 , 7 , 8 , 9 , 1 , 5 }; int n = arr . size (); quickSort (arr , 0 , n - 1); for (int i = 0 ; i < n ; i ++) { cout << arr [i] << " " ; } return 0 ; } C #include <stdio.h> void swap (int * a , int * b); // partition function int partition (int arr [], int low , int high) { // Choose the pivot int pivot = arr [high]; // Index of smaller element and indicates // the right position of pivot found so far int i = low - 1 ; // Traverse arr[low..high] and move all smaller // elements to the left side. Elements from low to // i are smaller after every iteration for (int j = low ; j <= high - 1 ; j ++) { if (arr [j] < pivot) { i ++ ; swap (& arr [i], & arr [j]); } } // Move pivot after smaller elements and // return its position swap (& arr [i + 1], & arr [high]); return i + 1 ; } // The QuickSort function implementation void quickSort (int arr [], int low , int high) { if (low < high) { // pi is the partition return index of pivot int pi = partition (arr , low , high); // recursion calls for smaller elements // and greater or equals elements quickSort (arr , low , pi - 1); quickSort (arr , pi + 1 , high); } } void swap (int * a , int * b) { int t = * a ; * a = * b ; * b = t ; } int main () { int arr [] = { 10 , 7 , 8 , 9 , 1 , 5 }; int n = sizeof (arr) / sizeof (arr [0]); quickSort (arr , 0 , n - 1); for (int i = 0 ; i < n ; i ++) { printf ("%d " , arr [i]); } return 0 ; } Java import java.util.Arrays ; class GfG { // partition function static int partition (int [] arr , int low , int high) { // choose the pivot int pivot = arr [high] ; // index of smaller element and indicates // the right position of pivot found so far int i = low - 1 ; // traverse arr[low..high] and move all smaller // elements to the left side. Elements from low to // i are smaller after every iteration for (int j = low ; j <= high - 1 ; j ++) { if (arr [j] < pivot) { i ++ ; swap (arr , i , j); } } // Move pivot after smaller elements and // return its position swap (arr , i + 1 , high); return i + 1 ; } // swap function static void swap (int [] arr , int i , int j) { int temp = arr [i] ; arr [i] = arr [j] ; arr [j] = temp ; } // the QuickSort function implementation static void quickSort (int [] arr , int low , int high) { if (low < high) { // pi is the partition return index of pivot int pi = partition (arr , low , high); // recursion calls for smaller elements // and greater or equals elements quickSort (arr , low , pi - 1); quickSort (arr , pi + 1 , high); } } public static void main (String [] args) { int [] arr = { 10 , 7 , 8 , 9 , 1 , 5 }; int n = arr . length ; quickSort (arr , 0 , n - 1); for (int val : arr) { System . out . print (val + " "); } } } Python # partition function def partition (arr , low , high): # choose the pivot pivot = arr [high] # index of smaller element and indicates # the right position of pivot found so far i = low - 1 # traverse arr[low..high] and move all smaller # elements to the left side. Elements from low to # i are smaller after every iteration for j in range (low , high): if arr [j] < pivot : i += 1 swap (arr , i , j) # move pivot after smaller elements and # return its position swap (arr , i + 1 , high) return i + 1 # swap function def swap (arr , i , j): arr [i], arr [j] = arr [j], arr [i] # the QuickSort function implementation def quickSort (arr , low , high): if low < high : # pi is the partition return index of pivot pi = partition (arr , low , high) # recursion calls for smaller elements # and greater or equals elements quickSort (arr , low , pi - 1) quickSort (arr , pi + 1 , high) if __name__ == "__main__" : arr = [10 , 7 , 8 , 9 , 1 , 5] n = len (arr) quickSort (arr , 0 , n - 1) for val in arr : print (val , end = " ") C# using System ; class GfG { // partition function static int partition (int [] arr , int low , int high) { // choose the pivot int pivot = arr [high]; // index of smaller element and indicates // the right position of pivot found so far int i = low - 1 ; // traverse arr[low..high] and move all smaller // elements to the left side. Elements from low to // i are smaller after every iteration for (int j = low ; j <= high - 1 ; j ++) { if (arr [j] < pivot) { i ++ ; swap (arr , i , j); } } // move pivot after smaller elements and // return its position swap (arr , i + 1 , high); return i + 1 ; } // swap function static void swap (int [] arr , int i , int j) { int temp = arr [i]; arr [i] = arr [j]; arr [j] = temp ; } // The QuickSort function implementation static void quickSort (int [] arr , int low , int high) { if (low < high) { // pi is the partition return index of pivot int pi = partition (arr , low , high); // recursion calls for smaller elements // and greater or equals elements quickSort (arr , low , pi - 1); quickSort (arr , pi + 1 , high); } } static void Main (string [] args) { int [] arr = { 10 , 7 , 8 , 9 , 1 , 5 }; int n = arr . Length ; quickSort (arr , 0 , n - 1); foreach (int val in arr) { Console . Write (val + " "); } } } JavaScript // partition function function partition (arr , low , high) { // choose the pivot let pivot = arr [high]; // index of smaller element and indicates // the right position of pivot found so far let i = low - 1 ; // traverse arr[low..high] and move all smaller // elements to the left side. Elements from low to // i are smaller after every iteration for (let j = low ; j <= high - 1 ; j ++) { if (arr [j] < pivot) { i ++ ; swap (arr , i , j); } } // move pivot after smaller elements and // return its position swap (arr , i + 1 , high); return i + 1 ; } // swap function function swap (arr , i , j) { let temp = arr [i]; arr [i] = arr [j]; arr [j] = temp ; } // the QuickSort function implementation function quickSort (arr , low , high) { if (low < high) { // pi is the partition return index of pivot let pi = partition (arr , low , high); // recursion calls for smaller elements // and greater or equals elements quickSort (arr , low , pi - 1); quickSort (arr , pi + 1 , high); } } // Driver Code let arr = [10 , 7 , 8 , 9 , 1 , 5]; let n = arr . length ; // call QuickSort on the entire array quickSort (arr , 0 , n - 1); for (let i = 0 ; i < arr . length ; i ++) { process . stdout . write (arr [i] + " "); } Output 1 5 7 8 9 10 Complexity Analysis of Quick Sort Time Complexity: Best Case: (Ω(n log n)), Occurs when the

pivot element divides the array into two equal halves. Average Case ($\theta(n \log n)$), On average, the pivot divides the array into two parts, but not necessarily equal. Worst Case: ($O(n^2)$), Occurs when the smallest or largest element is always chosen as the pivot (e.g., sorted arrays). Auxiliary Space: Worst-case scenario: O(n) due to unbalanced partitioning leading to a skewed recursion tree requiring a call stack of size O(n). Best-case scenario: O(log n) as a result of balanced partitioning leading to a balanced recursion tree with a call stack of size O(log n). Please refer Time and Space Complexity Analysis of Quick Sort for more details. Advantages of Quick Sort It is a divide-and-conquer algorithm that makes it easier to solve problems. It is efficient on large data sets. It has a low overhead, as it only requires a small amount of memory to function. It is Cache Friendly as we work on the same array to sort and do not copy data to any auxiliary array. Fastest general purpose algorithm for large data when stability is not required. It is tail recursive and hence all the tail call optimization can be done. Disadvantages of Quick Sort It has a worst-case time complexity of O(n 2), which occurs when the pivot is chosen poorly. It is not a good choice for small data sets. It is not a stable sort, meaning that if two elements have the same key, their relative order will not be preserved in the sorted output in case of quick sort, because here we are swapping elements according to the pivot's position (without considering their original positions). Applications of Quick Sort Sorting large datasets efficiently in memory. Used in library sort functions (like C++ std::sort and Java Arrays.sort for primitives). Arranging records in databases for faster searching. Preprocessing step in algorithms requiring sorted input (e.g., binary search, two-pointer techniques). Finding the kth smallest/largest element using Quickselect (a variant of quicksort). Sorting arrays of objects based on multiple keys (custom comparators). Data compression algorithms (like Huffman coding preprocessing). Graphics and computational geometry (e.g., convex hull algorithms). Please refer Application of Quicksort for more details. Comment Article Tags: Article Tags: Divide and Conquer Sorting DSA Adobe Qualcomm Samsung Goldman Sachs SAP Labs Target Corporation HSBC Quick Sort DSA Tutorials + 8 More