Merge Sort - GeeksforGeeks

Source: https://www.geeksforgeeks.org/merge-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 Merge Sort Last Updated : 3 Oct, 2025 Merge sort is a popular sorting algorithm known for its efficiency and stability. It follows the Divide and Conquer approach. It works by recursively dividing the input array into two halves, recursively sorting the two halves and finally merging them back together to obtain the sorted array. Here's a step-by-step explanation of how merge sort works: Divide: Divide the list or array recursively into two halves until it can no more be divided. Conquer: Each subarray is sorted individually using the merge sort algorithm. Merge: The sorted subarrays are merged back together in sorted order. The process continues until all elements from both subarrays have been merged. Let's sort the array or list [38, 27, 43, 10] using Merge Sort Let's look at the working of above example: Divide: [38, 27, 43, 10] is divided into [38, 27] and [43, 10] . [38, 27] is divided into [38] and [27] . [43, 10] is divided into [43] and [10] . Conquer: [38] is already sorted. [27] is already sorted. [43] is already sorted. [10] is already sorted. Merge: Merge [38] and [27] to get [27, 38] . Merge [43] and [10] to get [10,43] . Merge [27, 38] and [10,43] to get the final sorted list [10, 27, 38, 43] Therefore, the sorted list is [10, 27, 38, 43] . Try it on GfG Practice C++ #include <iostream> #include <vector> using namespace std ; // Merges two subarrays of arr[]. // First subarray is arr[left..mid] // Second subarray is arr[mid+1..right] void merge (vector < int >& arr , int left , int mid , int right){ int n1 = mid - left + 1 ; int n2 = right - mid ; // Create temp vectors vector < int > L (n1), R (n2); // Copy data to temp vectors L[] and R[] for (int i = 0 ; i < n1 ; i ++) L [i] = arr [left + i]; for (int j = 0 ; j < n2 ; j ++) R [j] = arr [mid + 1 + j]; int i = 0 , j = 0 ; int k = left ; // Merge the temp vectors back // into arr[left..right] while (i < n1 && j < n2) { if (L [i] <= R [j]) { arr [k] = L [i]; i ++ ; } else { arr [k] = R [j]; j ++ ; } k ++ ; } // Copy the remaining elements of L[], // if there are any while (i < n1) { arr [k] = L [i]; i ++ ; k ++ ; } // Copy the remaining elements of R[], // if there are any while (j < n2) { arr [k] = R [j]; j ++ ; k ++ ; } } // begin is for left index and end is right index // of the sub-array of arr to be sorted void mergeSort (vector < int >& arr , int left , int right){ if (left >= right) return ; int mid = left + (right - left) / 2 ; mergeSort (arr , left , mid); mergeSort (arr , mid + 1 , right); merge (arr , left , mid , right); } // Driver code int main (){ vector < int > arr = { 38 , 27 , 43 , 10 }; int n = arr . size (); mergeSort (arr , 0 , n - 1); for (int i = 0 ; i < arr . size (); i ++) cout << arr [i] << " " ; cout << endl ; return 0 ; } C #include <stdio.h> #include <stdlib.h> // Merges two subarrays of arr[]. // First subarray is arr[l..m] // Second subarray is arr[m+1..r] void merge (int arr [], int l , int m , int r){ int i , j , k ; int n1 = m - l + 1 ; int n2 = r - m ; // Create temp arrays int L [n1], R [n2]; // Copy data to temp arrays L[] and R[] for (i = 0 ; i < n1 ; i ++) L [i] = arr [l + i]; for (j = 0 ; j < n2 ; j ++) R [j] = arr [m + 1 + j]; // Merge the temp arrays back into arr[l..r i = 0 ; j = 0 ; k = l ; while (i < n1 && j < n2) { if (L [i] <= R [j]) { arr [k] = L [i]; i ++ ; } else { arr [k] = R [j]; j ++ ; } k ++ ; } // Copy the remaining elements of L[], // if there are any while (i < n1) { arr [k] = L [i]; i ++ ; k ++ ; } // Copy the remaining elements of R[], // if there are any while (j < n2) { arr [k] = R [j]; j ++ ; k ++ ; } } // l is for left index and r is right index of the // sub-array of arr to be sorted void mergeSort (int arr [], int l , int r){ if (l < r) { int m = l + (r - l) / 2 ; // Sort first and second halves mergeSort (arr , l , m); mergeSort (arr , m + 1 , r); merge (arr , l , m , r); } } // Driver code int main (){ int arr [] = { 38 , 27 , 43 , 10 }; int arr_size = sizeof (arr) / sizeof (arr [0]); mergeSort (arr , 0 , arr_size - 1); int i ; for (i = 0 ; i < arr_size ; i ++) printf ("%d " , arr [i]); printf (" \n "); return 0 ; } Java import java.io.* ; class GfG { // Merges two subarrays of arr[]. // First subarray is arr[l..m] // Second subarray is arr[m+1..r] static void merge (int arr [] , int l , int m , int r){ // Find sizes of two subarrays to be merged int n1 = m - l + 1 ; int n2 = r - m ; // Create temp arrays int L [] = new int [n1] ; int R [] = new int [n2] ; // Copy data to temp arrays for (int i

= 0 ; i < n1 ; ++ i) L [i] = arr [l + i] ; for (int j = 0 ; j < n2 ; ++ j) R [j] = arr [m + 1 + j] ; // Merge the temp arrays // Initial indices of first and second subarrays int i = 0 , j = 0 ; // Initial index of merged subarray array int k = l ; while (i < n1 && j < n2) { if (L [i] <= R [j]) { arr [k] = L [i] ; i ++ ; } else { arr [k] = R [j] ; j ++ ; } k ++ ; } // Copy remaining elements of L[] if any while (i < n1) { arr [k] = L [i] ; i ++ ; k ++ ; } // Copy remaining elements of R[] if any while (j < n2) { arr [k] = R [j] ; j ++ ; k ++ ; } } // Main function that sorts arr[l..r] using // merge() static void mergeSort (int arr [] , int l , int r){ if (l < r) { // Find the middle point int m = l + (r - l) / 2 ; // Sort first and second halves mergeSort (arr , l , m); mergeSort (arr , m + 1 , r); // Merge the sorted halves merge (arr , l , m , r); } } // Driver code public static void main (String args []){ int arr [] = { 38 , 27 , 43 , 10 }; mergeSort (arr , 0 , arr . length - 1); int n = arr . length ; for (int i = 0 ; i < n ; ++ i) System . out . print (arr [i] + " "); System . out . println (); } } Python def merge (arr , left , mid , right): n1 = mid - left + 1 n2 = right - mid # Create temp arrays L = [0] * n1 R = [0] * n2 # Copy data to temp arrays L[] and R[] for i in range (n1): L [i] = arr [left + i] for j in range (n2): R [j] = arr [mid + 1 + j] i = 0 j = 0 k = left # Merge the temp arrays back # into arr[left..right] while i < n1 and j < n2 : if L [i] <= R [j]: arr [k] = L [i] i += 1 else : arr [k] = R [j] j += 1 k += 1 # Copy the remaining elements of L[], # if there are any while i < n1 : arr [k] = L [i] i += 1 k += 1 # Copy the remaining elements of R[], # if there are any while j < n2 : arr [k] = R [j] j += 1 k += 1 def mergeSort (arr , left , right): if left < right : mid = (left + right) // 2 mergeSort (arr , left , mid) mergeSort (arr , mid + 1 , right) merge (arr , left , mid , right) # Driver code if __name__ == "__main__" : arr = [38 , 27 , 43 , 10] mergeSort (arr , 0 , len (arr) - 1) for i in arr : print (i , end = " ") print () C# using System ; class GfG { // Merges two subarrays of arr[]. // First subarray is arr[l..m] // Second subarray is arr[m+1..r] static void merge (int [] arr , int l , int m , int r){ // Find sizes of two // subarrays to be merged int n1 = m - l + 1 ; int n2 = r - m ; // Create temp arrays int [] L = new int [n1]; int [] R = new int [n2]; int i , j ; // Copy data to temp arrays for (i = 0 ; i < n1 ; ++ i) L [i] = arr [l + i]; for (j = 0 ; j < n2 ; ++ j) R [j] = arr [m + 1 + j]; // Merge the temp arrays // Initial indexes of first // and second subarrays i = 0 ; j = 0 ; // Initial index of merged // subarray array int k = l ; while (i < n1 && j < n2) { if (L [i] <= R [j]) { arr [k] = L [i]; i ++ ; } else { arr [k] = R [j]; j ++ ; } k ++ ; } // Copy remaining elements // of L[] if any while (i < n1) { arr [k] = L [i]; i ++ ; k ++ ; } // Copy remaining elements // of R[] if any while (j < n2) { arr [k] = R [j]; j ++ ; k ++ ; } } // Main function that sorts arr[l..r] using merge() static void mergeSort (int [] arr , int l , int r){ if (l < r) { // Find the middle point int m = l + (r - l) / 2 ; // Sort first and second halves mergeSort (arr , l , m); mergeSort (arr , m + 1 , r); // Merge the sorted halves merge (arr , l , m , r); } } // Driver code public static void Main (String [] args){ int [] arr = { 38 , 27 , 43 , 10 }; mergeSort (arr , 0 , arr . Length - 1); int n = arr . Length ; for (int i = 0 ; i < n ; ++ i) Console . Write (arr [i] + " "); Console . WriteLine (); } } JavaScript function merge (arr , left , mid , right) { const n1 = mid - left + 1 ; const n2 = right - mid ; // Create temp arrays const L = new Array (n1); const R = new Array (n2); // Copy data to temp arrays L[] and R[] for (let i = 0 ; i < n1 ; i ++) L [i] = arr [left + i]; for (let j = 0 ; j < n2 ; j ++) R [j] = arr [mid + 1 + j]; let i = 0 , j = 0 ; let k = left ; // Merge the temp arrays back into arr[left..right] while (i < n1 && j < n2) { if (L [i] <= R [j]) { arr [k] = L [i]; i ++ ; } else { arr [k] = R [j]; j ++ ; } k ++ ; } // Copy the remaining elements of L[], if there are any while (i < n1) { arr [k] = L [i]; i ++ ; k ++ ; } // Copy the remaining elements of R[], if there are any while (j < n2) { arr [k] = R [j]; j ++ ; k ++ ; } } function mergeSort (arr , left , right) { if (left >= right) return ; const mid = Math . floor (left + (right - left) / 2); mergeSort (arr , left , mid); mergeSort (arr , mid + 1 , right); merge (arr , left , mid , right); } // Driver code const arr = [38 , 27 , 43 , 10]; mergeSort (arr , 0 , arr . length - 1); console . log (arr . join (" ")); Output 10 27 38 43 Recurrence Relation of Merge Sort The recurrence relation of merge sort is: $T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 2T\left(\frac{n}{2}\right) + \Theta(n) & \text{if } n > 1 \end{cases}$ $T(n)$ Represents the total time time taken by the algorithm to sort an array of size n. $2T(n/2)$ represents time taken by the algorithm to recursively sort the two halves of the array. Since each half has n/2 elements, we have two recursive calls with input size as (n/2). $O(n)$ represents the time taken to merge the two sorted halves Complexity Analysis of Merge Sort Time Complexity: Best Case: O(n log n), When the array is already sorted or nearly sorted. Average Case: O(n log n), When the array is randomly ordered. Worst Case: O(n log n), When the array is sorted in reverse order. Auxiliary Space: O(n), Additional space is required for the temporary array used during merging. Applications of Merge Sort: Sorting large datasets External sorting (when the dataset is too large to fit in memory) Used to solve problems like Inversion counting , Count Smaller on Right & Surpasser Count Merge Sort and its variations are used in library methods of programming languages. Its variation TimSort is used in Python, Java Android and Swift. The main reason why it is preferred to sort non-primitive types is stability which is not there in QuickSort. Arrays.sort in Java uses QuickSort while Collections.sort uses MergeSort. It is a preferred algorithm for

sorting Linked lists. It can be easily parallelized as we can independently sort subarrays and then merge. The merge function of merge sort to efficiently solve the problems like union and intersection of two sorted arrays . Advantages and Disadvantages of Merge Sort Advantages Stability : Merge sort is a stable sorting algorithm, which means it maintains the relative order of equal elements in the input array. Guaranteed worst-case performance: Merge sort has a worst-case time complexity of O(N logN) , which means it performs well even on large datasets. Simple to implement: The divide-and-conquer approach is straightforward. Naturally Parallel : We independently merge subarrays that makes it suitable for parallel processing. Disadvantages Space complexity: Merge sort requires additional memory to store the merged sub-arrays during the sorting process. Not in-place: Merge sort is not an in-place sorting algorithm, which means it requires additional memory to store the sorted data. This can be a disadvantage in applications where memory usage is a concern. Merge Sort is Slower than QuickSort in general as QuickSort is more cache friendly because it works in-place. Quick Links: Merge Sort Based Coding Questions Bottom up (or Iterative) Merge Sort Sorting Interview Questions Quiz on Merge Sort Comment Article Tags: Article Tags: Divide and Conquer Sorting DSA Microsoft Amazon Oracle Qualcomm Goldman Sachs Snapdeal Paytm Target Corporation Grofers Boomerang Commerce Merge Sort + 10 More