Counting Sort - GeeksforGeeks

Source: https://www.geeksforgeeks.org/counting-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 Counting Sort Last Updated : 29 Sep, 2025 Counting Sort is a non-comparison-based sorting algorithm. It is particularly efficient when the range of input values is small compared to the number of elements to be sorted. The basic idea behind Counting Sort is to count the frequency of each distinct element in the input array and use that information to place the elements in their correct sorted positions. It works well when the range of input elements is small and comparable to the size of the array. For example, for input [1, 4, 0, 2, 1, 1], the size of array is 6 and range of elements is from 0 to 4 If range of input array is of order more than n Log n where n is size of the array, then we can better sort the array using a standard comparison based sorting algorithm like Merge Sort. Counting Sort Algorithm Declare a count array cntArr[] of size max(arr[])+1 and initialize it with 0 . Traverse input array arr[] and map each element of arr[] as an index of cntArr[] array, i.e., execute cntArr[arr[i]]++ for 0 <= i < N . Calculate the prefix sum at every index of cntArr[] . Create an array ans[] of size N . Traverse array arr[] from end and update ans[cntArr[arr[i]] - 1] = arr[i] . Also, update cntArr [arr[i]] = cntArr[arr[i]]- - . Why do we compute Prefix Sums? We could simply count occurrences of all elements and one by one put them in the output array, but we computer prefix sums to achieve stability in the algorithm. Note that, after building the prefix sum cntArr[], we traverse the array from right end to ensure that the last occurrence moves to the last correct position in the sorted array. Working of Counting Sort Try it on GfG Practice C++ #include <iostream> #include <vector> using namespace std ; vector < int > countsort (vector < int >& arr) { int n = arr . size (); // find the maximum element int maxval = 0 ; for (int i = 0 ; i < n ; i ++) maxval = max (maxval , arr [i]); // create and initialize cntArr array vector < int > cntArr (maxval + 1 , 0); // count frequency of each element for (int i = 0 ; i < n ; i ++) cntArr [arr [i]] ++ ; // compute prefix sum for (int i = 1 ; i <= maxval ; i ++) cntArr [i] += cntArr [i - 1]; // build output array vector < int > ans (n); for (int i = n - 1 ; i >= 0 ; i --) { ans [cntArr [arr [i]] - 1] = arr [i]; cntArr [arr [i]] -- ; } return ans ; } int main () { vector < int > arr = { 2 , 5 , 3 , 0 , 2 , 3 , 0 , 3 }; vector < int > ans = countsort (arr); for (int x : ans) cout << x << " " ; return 0 ; } C #include <stdio.h> #include <stdlib.h> void countsort (int arr [], int n) { int maxval = 0 ; // Find the maximum element for (int i = 0 ; i < n ; i ++) { if (arr [i] > maxval) { maxval = arr [i]; } } // Create and initialize cntArr array int * cntArr = (int *) calloc (maxval + 1 , sizeof (int)); // Count frequency of each element for (int i = 0 ; i < n ; i ++) { cntArr [arr [i]] ++ ; } // Compute prefix sum for (int i = 1 ; i <= maxval ; i ++) { cntArr [i] += cntArr [i - 1]; } // Build output array int * ans = (int *) malloc (n * sizeof (int)); for (int i = n - 1 ; i >= 0 ; i --) { ans [cntArr [arr [i]] - 1] = arr [i]; cntArr [arr [i]] -- ; } // Copy sorted elements back to arr[] for (int i = 0 ; i < n ; i ++) { arr [i] = ans [i]; } // Free dynamically allocated memory free (cntArr); free (ans); } // Driver code int main () { int arr [] = { 2 , 5 , 3 , 0 , 2 , 3 , 0 , 3 }; int n = sizeof (arr) / sizeof (arr [0]); // Sorting the array countsort (arr , n); // Printing the sorted array for (int i = 0 ; i < n ; i ++) { printf ("%d " , arr [i]); } return 0 ; } Java import java.util.Arrays ; public class CountingSort { public static int [] countSort (int [] arr) { int n = arr . length ; if (n == 0) { return new int [0] ; } // find the maximum element int maxVal = arr [0] ; for (int i = 1 ; i < n ; i ++) { if (arr [i] > maxVal) { maxVal = arr [i] ; } } // create and initialize cntArr array int [] cntArr = new int [maxVal + 1] ; for (int i = 0 ; i <= maxVal ; i ++) { cntArr [i] = 0 ; } // count frequency of each element for (int i = 0 ; i < n ; i ++) { cntArr [arr [i]]++ ; } // compute prefix sum for (int i = 1 ; i <= maxVal ; i ++) { cntArr [i] += cntArr [i - 1] ; } // build output array int [] ans = new int [n] ; for (int i = n - 1 ; i >= 0 ; i --) { int v = arr [i] ; ans [cntArr [v] - 1] = v ; cntArr [v]-- ; } return ans ; } public static void main (String [] args) { int [] arr = { 2 , 5 , 3 , 0 , 2 , 3 , 0 , 3 }; int [] ans = countSort (arr); System . out .

println (Arrays . toString (ans)); } } Python def count_sort (arr): if not arr : return [] n = len (arr) maxval = max (arr) # create and initialize cntArr cntArr = [0] * (maxval + 1) # count frequency of each element for v in arr : cntArr [v] += 1 # compute prefix sums for i in range (1 , maxval + 1): cntArr [i] += cntArr [i - 1] # build output array ans = [0] * n # iterate in reverse to keep it stable for i in range (n - 1 , - 1 , - 1): v = arr [i] ans [cntArr [v] - 1] = v cntArr [v] -= 1 return ans if __name__ == "__main__" : arr = [2 , 5 , 3 , 0 , 2 , 3 , 0 , 3] ans = count_sort (arr) print (ans) C# using System ; class CountingSort { public static int [] CountSort (int [] arr) { int n = arr . Length ; if (n == 0) { return new int [0]; } // find maximum int maxVal = arr [0]; foreach (int v in arr) { if (v > maxVal) maxVal = v ; } // create and initialize cntArr int [] cntArr = new int [maxVal + 1]; for (int i = 0 ; i <= maxVal ; i ++) { cntArr [i] = 0 ; } // count frequency foreach (int v in arr) { cntArr [v] ++ ; } // prefix sums for (int i = 1 ; i <= maxVal ; i ++) { cntArr [i] += cntArr [i - 1]; } // build output int [] ans = new int [n]; for (int i = n - 1 ; i >= 0 ; i --) { int v = arr [i]; ans [cntArr [v] - 1] = v ; cntArr [v] -- ; } return ans ; } static void Main (string [] args) { int [] arr = { 2 , 5 , 3 , 0 , 2 , 3 , 0 , 3 }; int [] ans = CountSort (arr); Console . WriteLine (string . Join (", " , ans)); } } JavaScript function countSort (arr) { if (arr . length === 0) { return []; } const n = arr . length ; const maxVal = Math . max (... arr); // create and initialize cntArr const cntArr = new Array (maxVal + 1). fill (0); // count frequency of each element for (let v of arr) { cntArr [v] ++ ; } // compute prefix sums for (let i = 1 ; i <= maxVal ; i ++) { cntArr [i] += cntArr [i - 1]; } // build output array const ans = new Array (n); for (let i = n - 1 ; i >= 0 ; i --) { const v = arr [i]; ans [cntArr [v] - 1] = v ; cntArr [v] -- ; } return ans ; } // Example usage: const arr = [2 , 5 , 3 , 0 , 2 , 3 , 0 , 3]; const ans = countSort (arr); console . log (ans); // [0, 0, 2, 2, 3, 3, 3, 5] Output 0 0 2 2 3 3 3 5 Complexity Analysis of Counting Sort: Time Complexity : O(N+M) in all cases, where N and M are the size of inputArray[] and countArray[] respectively. Auxiliary Space: O(N+M), where N and M are the space taken by outputArray[] and countArray[] respectively. Advantage, of Counting Sort: Counting sort generally performs faster than all comparison-based sorting algorithms, such as merge sort and quicksort, if the range of input is of the order of the number of input. Stable Algorithm Disadvantage of Counting Sort: Does not work on decimal values. Inefficient if the range of values to be sorted is very large. Not an In-place sorting algorithm, It uses extra space for sorting the array elements. Applications of Counting Sort: It is a commonly used algorithm for the cases where we have limited range items. For example, sort students by grades, sort a events by time, days, months, years, etc It is used as a subroutine in Radix Sort The idea of counting sort is used in Bucket Sort to divide elements into different buckets. Comment Article Tags: Article Tags: Sorting DSA Samsung Snapdeal counting-sort + 1 More