TimSort - GeeksforGeeks

Source: https://www.geeksforgeeks.org/timsort/

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 TimSort Last Updated : 30 Jan, 2026 TimSort is a hybrid sorting algorithm that uses the ideas of Merge Sort and Insertion Sort . Used as the default sorting algorithm in Python (sorted(), list.sort()) and Java (from Java 7 onwards for Arrays.sort() on objects). The key idea behind Timsort is to identify small sorted segments of the array, called runs , and then merge them efficiently to form the fully sorted array. How Timsort Works Timsort works in three main steps by combining ideas from Merge Sort and Insertion Sort: Identify Runs: It scans the array to find small segments that are already sorted, called runs. If a run is in descending order, it’s reversed to make it ascending. Sort Small Runs: If a run is shorter than a fixed size (usually 32), it is sorted using Insertion Sort, which is fast for small or nearly sorted data. Merge Runs: Finally, Timsort merges these runs using rules that keep merging balanced and efficient, similar to Merge Sort, but optimized for real-world data. C++ //Driver Code Starts #include <iostream> #include <vector> #include <algorithm> using namespace std ; //Driver Code Ends const int minRUN = 32 ; // Calculate minimum run length (kept small here for demo) int calcMinRun (int n) { int r = 0 ; while (n >= minRUN) { r |= (n & 1); n >>= 1 ; } return n + r ; } // Insertion sort for small ranges void insertionSort (vector < int >& arr , int left , int right) { for (int i = left + 1 ; i <= right ; i ++) { int key = arr [i]; int j = i - 1 ; while (j >= left && arr [j] > key) { arr [j + 1] = arr [j]; j -- ; } arr [j + 1] = key ; } } // Merge two sorted subarrays [l..m] and [m+1..r] void merge (vector < int >& arr , int l , int m , int r) { vector < int > left (arr . begin () + l , arr . begin () + m + 1); vector < int > right (arr . begin () + m + 1 , arr . begin () + r + 1); int i = 0 , j = 0 , k = l ; while (i < left . size () && j < right . size ()) { if (left [i] <= right [j]) arr [k ++] = left [i ++]; else arr [k ++] = right [j ++]; } while (i < left . size ()) arr [k ++] = left [i ++]; while (j < right . size ()) arr [k ++] = right [j ++]; } // Detect ascending/descending run starting at index "start" int findRun (vector < int >& arr , int start , int n) { int end = start + 1 ; if (end == n) return end ; if (arr [end] < arr [start]) { // descending while (end < n && arr [end] < arr [end - 1]) end ++ ; reverse (arr . begin () + start , arr . begin () + end); } else { // ascending while (end < n && arr [end] >= arr [end - 1]) end ++ ; } return end ; } // Timsort main function void timsort (vector < int >& arr) { int n = arr . size (); int minRun = calcMinRun (n); vector < pair < int , int >> runs ; int i = 0 ; while (i < n) { int runEnd = findRun (arr , i , n); int runLen = runEnd - i ; if (runLen < minRun) { int end = min (i + minRun , n); insertionSort (arr , i , end - 1); runEnd = end ; } runs . push_back ({ i , runEnd }); i = runEnd ; while (runs . size () > 1) { int l1 = runs [runs . size () - 2]. first ; int r1 = runs [runs . size () - 2]. second ; int l2 = runs [runs . size () - 1]. first ; int r2 = runs [runs . size () - 1]. second ; int len1 = r1 - l1 ; int len2 = r2 - l2 ; if (len1 <= len2) { merge (arr , l1 , r1 - 1 , r2 - 1); runs . pop_back (); runs [runs . size () - 1] = { l1 , r2 }; } else break ; } } while (runs . size () > 1) { int l1 = runs [runs . size () - 2]. first ; int r1 = runs [runs . size () - 2]. second ; int l2 = runs [runs . size () - 1]. first ; int r2 = runs [runs . size () - 1]. second ; merge (arr , l1 , r1 - 1 , r2 - 1); runs . pop_back (); runs [runs . size () - 1] = { l1 , r2 }; } } //Driver Code Starts int main () { vector < int > arr = { 5 , 21 , 7 , 23 , 19 , 10 , 1 , 3 , 2 }; timsort (arr); for (int x : arr) cout << x << " " ; cout << endl ; } //Driver Code Ends Java //Driver Code Starts import java.util.Arrays ; import java.util.List ; import java.util.ArrayList ; class GFG { //Driver Code Ends static final int minRUN = 32 ; // Calculate minimum run length (kept small here for demo) static int calcMinRun (int n) { int r = 0 ; while (n >= minRUN) { r |= (n & 1); n >>= 1 ; } return n + r ; } // Insertion sort for small ranges static void insertionSort (int [] arr , int left , int right) { for (int i = left + 1 ; i <= right ; i ++) { int key = arr [i] ; int j = i - 1 ; while (j >= left && arr [j] > key) { arr [j + 1] = arr [j] ; j -- ; } arr [j + 1] = key ; } } // Merge two sorted subarrays [l..m] and [m+1..r] static void merge (int [] arr , int l , int m , int r)

```
{ int [] left = Arrays . copyOfRange ( arr , l , m + 1 ); int [] right = Arrays . copyOfRange ( arr , m + 1 , r + 1 ); int i = 0 , j = 0 , k = l ; while ( i < left . length && j < right . length ) { if ( left [ i ] <= right [ j ] ) arr [ k ++] = left [ i ++] ; else arr [ k ++] = right [ j ++] ; } while ( i < left . length ) arr [ k ++] = left [ i ++] ; while ( j < right . length ) arr [ k ++] = right [ j ++] ; } // Detect ascending/descending run starting at index "start" static int findRun ( int [] arr , int start , int n ) { int end = start + 1 ; if ( end == n ) return end ; // Determine direction if ( arr [ end ] < arr [ start ] ) { // descending while ( end < n && arr [ end ] < arr [ end - 1 ] ) end ++ ; reverse ( arr , start , end - 1 ); } else { // ascending while ( end < n && arr [ end ] >= arr [ end - 1 ] ) end ++ ; } return end ; } // Reverse subarray from l to r static void reverse ( int [] arr , int l , int r ) { while ( l < r ) { int temp = arr [ l ] ; arr [ l ] = arr [ r ] ; arr [ r ] = temp ; l ++ ; r -- ; } } // Timsort main function static void timsort ( int [] arr ) { int n = arr . length ; int minRun = calcMinRun ( n ); List < int []> runs = new ArrayList <> (); int i = 0 ; while ( i < n ) { int runEnd = findRun ( arr , i , n ); int runLen = runEnd - i ; // Extend short runs to minRun using insertion sort if ( runLen < minRun ) { int end = Math . min ( i + minRun , n ); insertionSort ( arr , i , end - 1 ); runEnd = end ; } runs . add ( new int [] { i , runEnd }); i = runEnd ; // Maintain merge balance while ( runs . size () > 1 ) { int [] run1 = runs . get ( runs . size () - 2 ); int [] run2 = runs . get ( runs . size () - 1 ); int l1 = run1 [ 0 ] , r1 = run1 [ 1 ] ; int l2 = run2 [ 0 ] , r2 = run2 [ 1 ] ; int len1 = r1 - l1 , len2 = r2 - l2 ; if ( len1 <= len2 ) { merge ( arr , l1 , r1 - 1 , r2 - 1 ); runs . remove ( runs . size () - 1 ); runs . set ( runs . size () - 1 , new int [] { l1 , r2 }); } else break ; } } // Merge remaining runs while ( runs . size () > 1 ) { int [] run1 = runs . get ( runs . size () - 2 ); int [] run2 = runs . get ( runs . size () - 1 ); int l1 = run1 [ 0 ] , r1 = run1 [ 1 ] ; int l2 = run2 [ 0 ] , r2 = run2 [ 1 ] ; merge ( arr , l1 , r1 - 1 , r2 - 1 ); runs . remove ( runs . size () - 1 ); runs . set ( runs . size () - 1 , new int [] { l1 , r2 }); } } //Driver Code Starts public static void main ( String [] args ) { int [] arr = { 5 , 21 , 7 , 23 , 19 , 10 , 1 , 3 , 2 }; timsort ( arr ); for ( int x : arr ) System . out . print ( x + " " ); System . out . println (); } } //Driver Code Ends Python minRUN = 32 # Calculate minimum run length (kept small here for demo) def calcMinRun ( n ): r = 0 while n >= minRUN : r |= n & 1 n >>= 1 return n + r # Insertion sort for small ranges def insertionSort ( arr , left , right ): for i in range ( left + 1 , right + 1 ): key = arr [ i ] j = i - 1 while j >= left and arr [ j ] > key : arr [ j + 1 ] = arr [ j ] j -= 1 arr [ j + 1 ] = key # Merge two sorted subarrays [l..m] and [m+1..r] def merge ( arr , l , m , r ): left = arr [ l : m + 1 ] right = arr [ m + 1 : r + 1 ] i = j = 0 k = l while i < len ( left ) and j < len ( right ): if left [ i ] <= right [ j ]: arr [ k ] = left [ i ] i += 1 else : arr [ k ] = right [ j ] j += 1 k += 1 while i < len ( left ): arr [ k ] = left [ i ] i += 1 k += 1 while j < len ( right ): arr [ k ] = right [ j ] j += 1 k += 1 # Detect ascending/descending run starting at index "start" def findRun ( arr , start , n ): end = start + 1 if end == n : return end if arr [ end ] < arr [ start ]: # descending while end < n and arr [ end ] < arr [ end - 1 ]: end += 1 arr [ start : end ] = reversed ( arr [ start : end ]) else : # ascending while end < n and arr [ end ] >= arr [ end - 1 ]: end += 1 return end # Timsort main function def timsort ( arr ): n = len ( arr ) minRun = calcMinRun ( n ) runs = [] i = 0 while i < n : runEnd = findRun ( arr , i , n ) runLen = runEnd - i if runLen < minRun : end = min ( i + minRun , n ) insertionSort ( arr , i , end - 1 ) runEnd = end runs . append (( i , runEnd )) i = runEnd while len ( runs ) > 1 : l1 , r1 = runs [ - 2 ] l2 , r2 = runs [ - 1 ] len1 , len2 = r1 - l1 , r2 - l2 if len1 <= len2 : merge ( arr , l1 , r1 - 1 , r2 - 1 ) runs . pop () runs [ - 1 ] = ( l1 , r2 ) else : break while len ( runs ) > 1 : l1 , r1 = runs [ - 2 ] l2 , r2 = runs [ - 1 ] merge ( arr , l1 , r1 - 1 , r2 - 1 ) runs . pop () runs [ - 1 ] = ( l1 , r2 ) #Driver Code Starts if __name__ == "__main__" : arr = [ 5 , 21 , 7 , 23 , 19 , 10 , 1 , 3 , 2 ] timsort ( arr ) print ( arr ) #Driver Code Ends C# //Driver Code Starts using System ; using System.Collections.Generic ; class GFG { const int minRUN = 32 ; //Driver Code Ends // Calculate minimum run length (kept small here for demo) static int CalcMinRun ( int n ) { int r = 0 ; while ( n >= minRUN ) { r |= ( n & 1 ); n >>= 1 ; } return n + r ; } // Insertion sort for small ranges static void InsertionSort ( int [] arr , int left , int right ) { for ( int i = left + 1 ; i <= right ; i ++ ) { int key = arr [ i ]; int j = i - 1 ; while ( j >= left && arr [ j ] > key ) { arr [ j + 1 ] = arr [ j ]; j -- ; } arr [ j + 1 ] = key ; } } // Merge two sorted subarrays [l..m] and [m+1..r] static void Merge ( int [] arr , int l , int m , int r ) { int [] left = new int [ m - l + 1 ]; int [] right = new int [ r - m ]; Array . Copy ( arr , l , left , 0 , left . Length ); Array . Copy ( arr , m + 1 , right , 0 , right . Length ); int i = 0 , j = 0 , k = l ; while ( i < left . Length && j < right . Length ) { if ( left [ i ] <= right [ j ]) arr [ k ++ ] = left [ i ++ ]; else arr [ k ++ ] = right [ j ++ ]; } while ( i < left . Length ) arr [ k ++ ] = left [ i ++ ]; while ( j < right . Length ) arr [ k ++ ] = right [ j ++ ]; } // Detect ascending/descending run starting at index "start" static int FindRun ( int [] arr , int start , int n ) { int end = start + 1 ; if ( end == n ) return end ; if ( arr [ end ] < arr [ start ]) { while ( end < n && arr [ end ] < arr [ end - 1 ]) end ++ ; Array . Reverse ( arr , start , end - start ); } else { while ( end < n && arr [ end ] >= arr [ end - 1 ]) end ++ ; } return end ; } // Timsort main function static void TimSort ( int [] arr ) { int n = arr . Length ; int minRun = CalcMinRun ( n ); List < Tuple < int , int >> runs = new List < Tuple < int , int >> (); int i = 0 ; while ( i < n ) { int runEnd = FindRun ( arr , i , n ); int runLen = runEnd - i ; if ( runLen < minRun ) { int end = Math . Min ( i + minRun , n ); InsertionSort ( arr , i , end - 1 ); runEnd = end ; } runs . Add ( Tuple . Create ( i ,
```

runEnd)); i = runEnd ; while (runs . Count > 1) { int l1 = runs [runs . Count - 2]. Item1 ; int r1 = runs [runs . Count - 2]. Item2 ; int l2 = runs [runs . Count - 1]. Item1 ; int r2 = runs [runs . Count - 1]. Item2 ; int len1 = r1 - l1 ; int len2 = r2 - l2 ; if (len1 <= len2) { Merge (arr , l1 , r1 - 1 , r2 - 1); runs . RemoveAt (runs . Count - 1); runs [runs . Count - 1] = Tuple . Create (l1 , r2); } else break ; } } while (runs . Count > 1) { int l1 = runs [runs . Count - 2]. Item1 ; int r1 = runs [runs . Count - 2]. Item2 ; int l2 = runs [runs . Count - 1]. Item1 ; int r2 = runs [runs . Count - 1]. Item2 ; Merge (arr , l1 , r1 - 1 , r2 - 1); runs . RemoveAt (runs . Count - 1); runs [runs . Count - 1] = Tuple . Create (l1 , r2); } } //Driver Code Starts static void Main () { int [] arr = { 5 , 21 , 7 , 23 , 19 , 10 , 1 , 3 , 2 }; TimSort (arr); Console . WriteLine (string . Join (" " , arr)); } } //Driver Code Ends JavaScript const minRUN = 32 ; // Calculate minimum run length (kept small here for demo) function calcMinRun (n) { let r = 0 ; while (n >= minRUN) { r |= n & 1 ; n >>= 1 ; } return n + r ; } // Insertion sort for small ranges function insertionSort (arr , left , right) { for (let i = left + 1 ; i <= right ; i ++) { let key = arr [i]; let j = i - 1 ; while (j >= left && arr [j] > key) { arr [j + 1] = arr [j]; j -- ; } arr [j + 1] = key ; } } // Merge two sorted subarrays [l..m] and [m+1..r] function merge (arr , l , m , r) { const left = arr . slice (l , m + 1); const right = arr . slice (m + 1 , r + 1); let i = 0 , j = 0 , k = l ; while (i < left . length && j < right . length) { if (left [i] <= right [j]) arr [k ++] = left [i ++]; else arr [k ++] = right [j ++]; } while (i < left . length) arr [k ++] = left [i ++]; while (j < right . length) arr [k ++] = right [j ++]; } // Detect ascending/descending run starting at index "start" function findRun (arr , start , n) { let end = start + 1 ; if (end === n) return end ; if (arr [end] < arr [start]) { // descending while (end < n && arr [end] < arr [end - 1]) end ++ ; let sub = arr . slice (start , end). reverse (); for (let i = start ; i < end ; i ++) arr [i] = sub [i - start]; } else { // ascending while (end < n && arr [end] >= arr [end - 1]) end ++ ; } return end ; } // Timsort main function function timsort (arr) { const n = arr . length ; const minRun = calcMinRun (n); const runs = []; let i = 0 ; while (i < n) { let runEnd = findRun (arr , i , n); let runLen = runEnd - i ; // Extend short runs to minRun using insertion sort if (runLen < minRun) { let end = Math . min (i + minRun , n); insertionSort (arr , i , end - 1); runEnd = end ; } runs . push ([i , runEnd]); i = runEnd ; while (runs . length > 1) { const [l1 , r1] = runs [runs . length - 2]; const [l2 , r2] = runs [runs . length - 1]; const len1 = r1 - l1 , len2 = r2 - l2 ; if (len1 <= len2) { merge (arr , l1 , r1 - 1 , r2 - 1); runs . pop (); runs [runs . length - 1] = [l1 , r2]; } else break ; } } while (runs . length > 1) { const [l1 , r1] = runs [runs . length - 2]; const [l2 , r2] = runs [runs . length - 1]; merge (arr , l1 , r1 - 1 , r2 - 1); runs . pop (); runs [runs . length - 1] = [l1 , r2]; } } // Example usage //Driver Code Starts let arr = [5 , 21 , 7 , 23 , 19 , 10 , 1 , 3 , 2]; timsort (arr); console . log (arr . join (" ")); //Driver Code Ends Output 1 2 3 5 7 10 19 21 23 Complexity Analysis of Tim Sort: Time Complexity: Best Case: $\Omega(n)$, when the array is already sorted. Average Case: $\Theta(n \log n)$, 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 merging runs. Stability: Yes, Tim Sort is a stable sorting algorithm (maintains the relative order of equal elements). Understanding Time Complexity For Tim Sort: Timsort's time complexity comes from the combination of two phases — sorting small runs using Insertion Sort and merging those runs using a Merge Sort like process. Sorting Part: Detecting the runs takes linear time, $O(n)$. Each run is then sorted using Insertion Sort, which takes $O(k^2)$ for a run of size k. Since there are about n/k such runs, the total cost for sorting all runs becomes $O(n*k)$, which simplifies to $O(n)$ because k is a small constant. Merging Part: Merging two runs of total length m takes $O(m)$ time. Every element in the array participates in each level of merging once, so each merge level costs $O(n)$. As runs double in size after every merge, the number of merge levels is approximately $\log(n/k)$. Hence, the total merging cost becomes $O(n \log(n/k))$, which simplifies to $O(n \log n)$. Combining both steps gives the overall time complexity: sorting runs $O(n)$ + merging runs $O(n \log n) = O(n \log n)$. Applications of Tim Sort: Used as the default sorting algorithm in Python (sorted(), list.sort()) and Java (from Java 7 onwards for Arrays.sort() on objects). Particularly effective for real-world datasets where data is often partially sorted. Suitable for sorting large datasets where stability is required. Performs well in database systems and search engines where maintaining order of equal keys matters. Used in Android, Swift, and other libraries due to its stability and efficiency. Advantages and Disadvantages of Tim Sort: Advantages: Stability: Tim Sort is a stable sorting algorithm, so equal elements maintain their original order. Adaptive: Performs better on partially sorted data, often achieving close to $O(n)$ performance. Guaranteed Worst-Case Performance: Always bounded by $O(n \log n)$, making it reliable. Practical Efficiency: Combines insertion sort (efficient on small runs) and merge sort (efficient on large data), making it faster for real-world use cases. Disadvantages: Space Complexity: Requires additional memory $O(n)$ during merging, making it not strictly in-place. Implementation Complexity: More complex to implement compared to simpler algorithms like Quick Sort or Merge Sort. Not Cache-Friendly: Like merge sort, it may be slower in low-memory systems compared to in-place algorithms like Quick Sort.

Complexity Comparison with Merge and Quick Sort: Algorithm Time Complexity Best Average Worst Quick Sort Ω(n*log(n)) θ(n*log(n)) O(n^2) Merge Sort Ω(n*log(n)) θ(n*log(n)) O(n*log(n)) Tim Sort Ω(n) θ(n*log(n)) O(n*log(n)) Comment Article Tags: Article Tags: Sorting DSA Merge Sort Insertion Sort