IntroSort or Introspective sort - GeeksforGeeks

Source: https://www.geeksforgeeks.org/introsort-or-introspective-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 IntroSort or Introspective sort Last Updated : 11 Jul, 2025 Introsort(Introspective sort) is a comparison based sort that consists of three sorting phases. They are Quicksort, Heapsort, and Insertion sort. Basic concepts of Introsort and the C++ code are available here The following section shows how the Introsort algorithm is formulated, after reviewing the pros and cons of the respective algorithms. Quicksort The Quicksort is an efficient sorting algorithm but has the worst-case performance of O(N ^ 2) comparisons with O(N) auxiliary space. This worst-case complexity depends on two phases of the Quicksort algorithm. 1. Choosing the pivot element 2. Recursion depth during the course of the algorithm Heapsort Heapsort has O(N log N) worst-case time complexity, which is much better than the worst case of Quicksort. So, is it evident that Heapsort is the best? No, the secret of Quicksort is that it does not swap already elements that are already in order, which is unnecessary, whereas with the Heapsort, even if all of the data is already sorted, the algorithm swaps all of the elements to order the array. Also, by choosing the optimal pivot, the worst-case of O(N ^ 2) can be avoided in quicksort. But, the swapping will pay more time in the case of Heapsort that is unavoidable. Hence, Quicksort outperforms Heapsort. The best things about Heapsort is that, if the recursion depth becomes too large like (log N), the worst case complexity would be still O(N log N). Mergesort The Mergesort has the worst case complexity only as O(N log N). Mergesort can work well on any type of data sets irrespective of its size whereas the Quicksort cannot work well with large data sets. But, Mergesort is not in-place whereas Quicksort is in-place, and that plays a vital role in here. Mergesort goes well with LinkedLists whereas Quicksort goes well with arrays. The locality of reference is better with Quicksort, whereas with Mergesort it is bad. So, for conventional purposes, having memory constraints in hand, Quicksort outperforms Mergesort. Insertion sort The main advantage of the insertion sort is its simplicity. It also exhibits a good performance when dealing with a small list. The insertion sort is an in-place sorting algorithm so the space requirement is minimal. The disadvantage of the insertion sort is that it does not perform as well as the other sorting algorithms when the size of the data gets larger. Here is how Introsort is formulated: Choosing the right sorting algorithm depends on the occasion where the sorting algorithm is being used. There are a good number of sorting algorithms in hand already that has pros and cons of its own. So, to get a better sorting algorithm, the solution is to tweak the existing algorithms and produce a new sorting algorithm that works better. There are a lot of hybrid algorithms, that outperforms the general sorting algorithms. One such is the Introsort. The best versions of Quicksort are competitive with both heap sort and merge sort on the vast majority of inputs. Rarely Quicksort has the worst case of O(N ^ 2) running time and O(N) stack usage. Both Heapsort and Mergesort have O(N log N) worst-case running time, together with a stack usage of O(1) for Heapsort and O(log N) for Mergesort respectively. Also, Insertion sort performs better than any of the above algorithms if the data set is small. Combining all the pros of the sorting algorithms, Introsort behaves based on the data set. If the number of elements in the input gets fewer, the Introsort performs Insertion sort for the input. Having the least number of comparisons(Quicksort) in mind, for splitting the array by finding the pivot element, Quicksort is used. Quoted earlier, the worst case of Quicksort is based on the two phases and here is how we can fix them. Choosing the pivot element: We can use either of median-of-3 concept or randomized pivot concept or middle as the pivot concept for finding the pivot element Recursion depth during the course of the algorithm: When the recursion depth gets higher, Introsort uses Heapsort as it has the definite upper bound of O(N log N). How does depthLimit work? depthLimit represents

maximum depth for recursion. It is typically chosen as log of length of input array (please refer below implementation). The idea is to ensure that the worst case time complexity remains O(N log N). Note that the worst-case time complexity of HeapSort is O(N log N). Why is Mergesort not used? As the arrays are being dealt with the in-place concept where Quicksort outperforms Mergesort, we are not using Mergesort. Can Introsort be applied everywhere? If the data won't fit in an array, Introsort cannot be used. Furthermore, like Quicksort and Heapsort, Introsort is not stable. When a stable sort is needed, Introsort cannot be applied. Is Introsort the only hybrid sorting algorithm? No. There are other hybrid sorting algorithms like Hybrid Mergesort, Tim sort, Insertion-Merge hybrid. Comparison of Heapsort, Insertion sort, Quicksort, Introsort while sorting 6000 elements(in milliseconds). Pseudocode: sort(A : array): depthLimit = 2xfloor(log(length(A))) introsort(A, depthLimit) introsort(A, depthLimit): n = length(A) if n<=16: insertionSort(A) if depthLimit == 0: heapsort(A) else: // using quick sort, the // partition point is found p = partition(A) introsort(A[0:p-1], depthLimit - 1) introsort(A[p+1:n], depthLimit - 1) Time Complexity: Worst-case performance: O(nlogn) (better than Quicksort) Average-case performance: O(nlogn) In the Quicksort phase, the pivot can either be chosen using the median-of-3 concept or last element of the array. For data that has a huge number of elements, median-of-3 concept slows down the running time of the Quicksort. In the example described below, the quicksort algorithm calculates the pivot element based on the median-of-3 concept. Example: C++ // C++ implementation of Introsort algorithm #include <bits/stdc++.h> using namespace std ; // A utility function to swap the values pointed by // the two pointers void swapValue (int * a , int * b) { int * temp = a ; a = b ; b = temp ; return ; } /* Function to sort an array using insertion sort*/ void InsertionSort (int arr [], int * begin , int * end) { // Get the left and the right index of the subarray // to be sorted int left = begin - arr ; int right = end - arr ; for (int i = left + 1 ; i <= right ; i ++) { int key = arr [i]; int j = i - 1 ; /* Move elements of arr[0..i-1], that are greater than key, to one position ahead of their current position */ while (j >= left && arr [j] > key) { arr [j + 1] = arr [j]; j = j - 1 ; } arr [j + 1] = key ; } return ; } // A function to partition the array and return // the partition point 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) { // increment index of smaller element i ++ ; swap (arr [i], arr [j]); } } swap (arr [i + 1], arr [high]); return (arr + i + 1); } // A function that find the middle of the // values pointed by the pointers a, b, c // and return that pointer int * MedianOfThree (int * a , int * b , int * c) { if (* a < * b && * b < * c) return (b); if (* a < * c && * c <= * b) return (c); if (* b <= * a && * a < * c) return (a); if (* b < * c && * c <= * a) return (c); if (* c <= * a && * a < * b) return (a); if (* c <= * b && * b <= * a) return (b); } // A Utility function to perform intro sort void IntrosortUtil (int arr [], int * begin , int * end , int depthLimit) { // Count the number of elements int size = end - begin ; // If partition size is low then do insertion sort if (size < 16) { InsertionSort (arr , begin , end); return ; } // If the depth is zero use heapsort if (depthLimit == 0) { make_heap (begin , end + 1); sort_heap (begin , end + 1); return ; } // Else use a median-of-three concept to // find a good pivot int * pivot = MedianOfThree (begin , begin + size / 2 , end); // Swap the values pointed by the two pointers swapValue (pivot , end); // Perform Quick Sort int * partitionPoint = Partition (arr , begin - arr , end - arr); IntrosortUtil (arr , begin , partitionPoint - 1 , depthLimit - 1); IntrosortUtil (arr , partitionPoint + 1 , end , depthLimit - 1); return ; } /* Implementation of introsort*/ void Introsort (int arr [], int * begin , int * end) { int depthLimit = 2 * log (end - begin); // Perform a recursive Introsort IntrosortUtil (arr , begin , end , depthLimit); return ; } // A utility function ot print an array of size n void printArray (int arr [], int n) { for (int i = 0 ; i < n ; i ++) cout << arr [i] << " \n " [i + 1 == n]; } // Driver program to test Introsort int main () { int arr [] = { 2 , 10 , 24 , 2 , 10 , 11 , 27 , 4 , 2 , 4 , 28 , 16 , 9 , 8 , 28 , 10 , 13 , 24 , 22 , 28 , 0 , 13 , 27 , 13 , 3 , 23 , 18 , 22 , 8 , 8 }; int n = sizeof (arr) / sizeof (arr [0]); // Pass the array, the pointer to the first element and // the pointer to the last element Introsort (arr , arr , arr + n - 1); printArray (arr , n); return (0); } Java // Java implementation of Introsort algorithm import java.io.IOException ; public class Introsort { // the actual data that has to be sorted private int a [] ; // the number of elements in the data private int n ; // Constructor to initialize the size // of the data Introsort (int n) { a = new int [n] ; this . n = 0 ; } // The utility function to insert the data private void dataAppend (int temp) { a [n] = temp ; n ++ ; } // The utility function to swap two elements private void swap (int i , int j) { int temp = a [i] ; a [i] = a [j] ; a [j] = temp ; } // To maxHeap a subtree rooted with node i which is // an index in a[]. heapN is size of heap private void maxHeap (int i , int heapN , int begin) { int temp = a [begin + i - 1] ; int child ; while (i <= heapN / 2) { child = 2 * i ; if (child < heapN && a [begin + child - 1] < a [begin + child]) child ++ ; if (temp >= a [begin + child - 1]) break ; a [begin + i - 1] = a [begin + child - 1] ; i = child ; } a [begin + i - 1] = temp ; } // Function to build the heap (rearranging the array) private void heapify (int begin , int end , int heapN) { for (int i = (heapN) / 2 ; i >= 1 ; i --) maxHeap (i , heapN ,

```
begin ); } // main function to do heapsort private void heapSort ( int begin , int end ) { int heapN = end - begin ; // Build heap (rearrange array) this . heapify ( begin , end , heapN ); // One by one extract an element from heap for ( int i = heapN ; i >= 1 ; i -- ) { // Move current root to end swap ( begin , begin + i ); // call maxHeap() on the reduced heap maxHeap ( 1 , i , begin ); } } // function that implements insertion sort private void insertionSort ( int left , int right ) { for ( int i = left ; i <= right ; i ++ ) { int key = a [ i ] ; int j = i ; // Move elements of arr[0..i-1], that are // greater than the key, to one position ahead // of their current position while ( j > left && a [ j - 1 ] > key ) { a [ j ] = a [ j - 1 ] ; j -- ; } a [ j ] = key ; } } // Function for finding the median of the three elements private int findPivot ( int a1 , int b1 , int c1 ) { int max = Math . max ( Math . max ( a [ a1 ] , a [ b1 ] ), a [ c1 ] ); int min = Math . min ( Math . min ( a [ a1 ] , a [ b1 ] ), a [ c1 ] ); int median = max ^ min ^ a [ a1 ] ^ a [ b1 ] ^ a [ c1 ] ; if ( median == a [ a1 ] ) return a1 ; if ( median == a [ b1 ] ) return b1 ; return c1 ; } // 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 greater elements to the right of the pivot private int partition ( int low , int high ) { // pivot int pivot = a [ high ] ; // Index of smaller element int i = ( low - 1 ); for ( int j = low ; j <= high - 1 ; j ++ ) { // If the current element is smaller // than or equal to the pivot if ( a [ j ] <= pivot ) { // increment index of smaller element i ++ ; swap ( i , j ); } } swap ( i + 1 , high ); return ( i + 1 ); } // The main function that implements Introsort // low --> Starting index, // high --> Ending index, // depthLimit --> recursion level private void sortDataUtil ( int begin , int end , int depthLimit ) { if ( end - begin > 16 ) { if ( depthLimit == 0 ) { // if the recursion limit is // occurred call heap sort this . heapSort ( begin , end ); return ; } depthLimit = depthLimit - 1 ; int pivot = findPivot ( begin , begin + (( end - begin ) / 2 ) + 1 , end ); swap ( pivot , end ); // p is partitioning index, // arr[p] is now at right place int p = partition ( begin , end ); // Separately sort elements before // partition and after partition sortDataUtil ( begin , p - 1 , depthLimit ); sortDataUtil ( p + 1 , end , depthLimit ); } else { // if the data set is small, // call insertion sort insertionSort ( begin , end ); } } // A utility function to begin the // Introsort module private void sortData () { // Initialise the depthLimit // as 2*log(length(data)) int depthLimit = ( int )( 2 * Math . floor ( Math . log ( n ) / Math . log ( 2 ))); this . sortDataUtil ( 0 , n - 1 , depthLimit ); } // A utility function to print the array data private void printData () { for ( int i = 0 ; i < n ; i ++ ) System . out . print ( a [ i ] + " " ); } // Driver code public static void main ( String args [] ) throws IOException { int [] inp = { 2 , 10 , 24 , 2 , 10 , 11 , 27 , 4 , 2 , 4 , 28 , 16 , 9 , 8 , 28 , 10 , 13 , 24 , 22 , 28 , 0 , 13 , 27 , 13 , 3 , 23 , 18 , 22 , 8 , 8 }; int n = inp . length ; Introsort introsort = new Introsort ( n ); for ( int i = 0 ; i < n ; i ++ ) { introsort . dataAppend ( inp [ i ] ); } introsort . sortData (); introsort . printData (); } } C# // C# implementation of // Introsort algorithm using System ; class Introsort { // the actual data that // has to be sorted public int [] a ; // the number of elements // in the data public int n ; // Constructor to initialize // the size of the data Introsort ( int n ) { a = new int [ n ]; this . n = 0 ; } // The utility function to // insert the data private void dataAppend ( int temp ) { a [ n ] = temp ; n ++ ; } // The utility function to // swap two elements private void swap ( int i , int j ) { int temp = a [ i ]; a [ i ] = a [ j ]; a [ j ] = temp ; } // To maxHeap a subtree rooted // with node i which is an index // in []a. heapN is size of heap private void maxHeap ( int i , int heapN , int begin ) { int temp = a [ begin + i - 1 ]; int child ; while ( i <= heapN / 2 ) { child = 2 * i ; if ( child < heapN && a [ begin + child - 1 ] < a [ begin + child ]) child ++ ; if ( temp >= a [ begin + child - 1 ]) break ; a [ begin + i - 1 ] = a [ begin + child - 1 ]; i = child ; } a [ begin + i - 1 ] = temp ; } // Function to build the // heap (rearranging the array) private void heapify ( int begin , int end , int heapN ) { for ( int i = ( heapN ) / 2 ; i >= 1 ; i -- ) maxHeap ( i , heapN , begin ); } // main function to do heapsort private void heapSort ( int begin , int end ) { int heapN = end - begin ; // Build heap (rearrange array) this . heapify ( begin , end , heapN ); // One by one extract an element // from heap for ( int i = heapN ; i >= 1 ; i -- ) { // Move current root to end swap ( begin , begin + i ); // call maxHeap() on the // reduced heap maxHeap ( 1 , i , begin ); } } // function that implements // insertion sort private void insertionSort ( int left , int right ) { for ( int i = left ; i <= right ; i ++ ) { int key = a [ i ]; int j = i ; // Move elements of arr[0..i-1], // that are greater than the key, // to one position ahead // of their current position while ( j > left && a [ j - 1 ] > key ) { a [ j ] = a [ j - 1 ]; j -- ; } a [ j ] = key ; } } // Function for finding the median // of the three elements private int findPivot ( int a1 , int b1 , int c1 ) { int max = Math . Max ( Math . Max ( a [ a1 ], a [ b1 ]), a [ c1 ]); int min = Math . Min ( Math . Min ( a [ a1 ], a [ b1 ]), a [ c1 ]); int median = max ^ min ^ a [ a1 ] ^ a [ b1 ] ^ a [ c1 ]; if ( median == a [ a1 ]) return a1 ; if ( median == a [ b1 ]) return b1 ; return c1 ; } // 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 greater elements to // the right of the pivot private int partition ( int low , int high ) { // pivot int pivot = a [ high ]; // Index of smaller element int i = ( low - 1 ); for ( int j = low ; j <= high - 1 ; j ++ ) { // If the current element // is smaller than or equal // to the pivot if ( a [ j ] <= pivot ) { // increment index of // smaller element i ++ ; swap ( i , j ); } } swap ( i + 1 , high ); return ( i
```

```
+ 1 ); } // The main function that implements // Introsort low --> Starting index, // high --> Ending index, depthLimit // --> recursion level private void sortDataUtil ( int begin , int end , int depthLimit ) { if ( end - begin > 16 ) { if ( depthLimit == 0 ) { // if the recursion limit is // occurred call heap sort this . heapSort ( begin , end ); return ; } depthLimit = depthLimit - 1 ; int pivot = findPivot ( begin , begin + (( end - begin ) / 2 ) + 1 , end ); swap ( pivot , end ); // p is partitioning index, // arr[p] is now at right place int p = partition ( begin , end ); // Separately sort elements // before partition and after // partition sortDataUtil ( begin , p - 1 , depthLimit ); sortDataUtil ( p + 1 , end , depthLimit ); } else { // if the data set is small, // call insertion sort insertionSort ( begin , end ); } } // A utility function to begin // the Introsort module private void sortData () { // Initialise the depthLimit // as 2*log(length(data)) int depthLimit = ( int )( 2 * Math . Floor ( Math . Log ( n ) / Math . Log ( 2 ))); this . sortDataUtil ( 0 , n - 1 , depthLimit ); } // A utility function to print // the array data private void printData () { for ( int i = 0 ; i < n ; i ++ ) Console . Write ( a [ i ] + " " ); } // Driver code public static void Main ( String [] args ) { int [] inp = { 2 , 10 , 24 , 2 , 10 , 11 , 27 , 4 , 2 , 4 , 28 , 16 , 9 , 8 , 28 , 10 , 13 , 24 , 22 , 28 , 0 , 13 , 27 , 13 , 3 , 23 , 18 , 22 , 8 , 8 }; int n = inp . Length ; Introsort introsort = new Introsort ( n ); for ( int i = 0 ; i < n ; i ++ ) { introsort . dataAppend ( inp [ i ]); } introsort . sortData (); introsort . printData (); } } // This code is contributed by Rajput-Ji JavaScript let arr = []; // Function to perform heapsort function heapsort () { const h = []; // Building the heap for ( const value of arr ) { h . push ( value ); } arr = []; // Extracting the sorted elements one by one while ( h . length > 0 ) { arr . push ( Math . min (... h )); h . splice ( h . indexOf ( Math . min (... h )), 1 ); } } // Function to perform insertion sort function insertionSort ( begin , end ) { const left = begin ; const right = end ; // Traverse through 1 to arr.length for ( let i = left + 1 ; i <= right ; i ++ ) { const key = arr [ i ]; let j = i - 1 ; // Move elements of arr[0..i-1] greater than key, to one position ahead of their current position while ( j >= left && arr [ j ] > key ) { arr [ j + 1 ] = arr [ j ]; j = j - 1 ; } arr [ j + 1 ] = key ; } } // Function to partition the array function partition ( low , high ) { const pivot = arr [ high ]; let i = low - 1 ; for ( let j = low ; j < high ; j ++ ) { if ( arr [ j ] <= pivot ) { i = i + 1 ; [ arr [ i ], arr [ j ]] = [ arr [ j ], arr [ i ]]; } } [ arr [ i + 1 ], arr [ high ]] = [ arr [ high ], arr [ i + 1 ]]; return i + 1 ; } // Function to find the median of three elements function medianOfThree ( a , b , d ) { const A = arr [ a ]; const B = arr [ b ]; const C = arr [ d ]; if (( A <= B && B <= C ) || ( C <= B && B <= A )) return b ; if (( B <= A && A <= C ) || ( C <= A && A <= B )) return a ; if (( B <= C && C <= A ) || ( A <= C && C <= B )) return d ; } // Main function to perform Introsort function introsortUtil ( begin , end , depthLimit ) { const size = end - begin + 1 ; if ( size < 16 ) { insertionSort ( begin , end ); return ; } if ( depthLimit === 0 ) { heapsort (); return ; } const pivot = medianOfThree ( begin , begin + Math . floor ( size / 2 ), end ); [ arr [ pivot ], arr [ end ]] = [ arr [ end ], arr [ pivot ]]; const partitionPoint = partition ( begin , end ); introsortUtil ( begin , partitionPoint - 1 , depthLimit - 1 ); introsortUtil ( partitionPoint + 1 , end , depthLimit - 1 ); } // Utility function to begin the Introsort module function introsort ( begin , end ) { const depthLimit = 2 * Math . floor ( Math . log2 ( end - begin + 1 )); introsortUtil ( begin , end , depthLimit ); } // Utility function to print the array data function printArr () { console . log ( arr . join ( ' ' )); // Modified to use join } // Main function function main () { arr = [ 2 , 10 , 24 , 2 , 10 , 11 , 27 , 4 , 2 , 4 , 28 , 16 , 9 , 8 , 28 , 10 , 13 , 24 , 22 , 28 , 0 , 13 , 27 , 13 , 3 , 23 , 18 , 22 , 8 , 8 ]; introsort ( 0 , arr . length - 1 ); printArr (); } main (); Python3 # Python implementation of Introsort algorithm import math import sys from heapq import heappush , heappop arr = [] # The main function to sort # an array of the given size # using heapsort algorithm def heapsort (): global arr h = [] # building the heap for value in arr : heappush ( h , value ) arr = [] # extracting the sorted elements one by one arr = arr + [ heappop ( h ) for i in range ( len ( h ))] # The main function to sort the data using # insertion sort algorithm def InsertionSort ( begin , end ): left = begin right = end # Traverse through 1 to len(arr) for i in range ( left + 1 , right + 1 ): key = arr [ i ] # Move elements of arr[0..i-1], that are # greater than key, to one position ahead # of their current position j = i - 1 while j >= left and arr [ j ] > key : arr [ j + 1 ] = arr [ j ] j = j - 1 arr [ j + 1 ] = key # 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 ( low , high ): global arr # pivot pivot = arr [ high ] # index of smaller element i = low - 1 for j in range ( low , high ): # If the current element is smaller than or # equal to the pivot if arr [ j ] <= pivot : # increment index of smaller element i = i + 1 ( arr [ i ], arr [ j ]) = ( arr [ j ], arr [ i ]) ( arr [ i + 1 ], arr [ high ]) = ( arr [ high ], arr [ i + 1 ]) return i + 1 # The function to find the median # of the three elements in # in the index a, b, d def MedianOfThree ( a , b , d ): global arr A = arr [ a ] B = arr [ b ] C = arr [ d ] if A <= B and B <= C : return b if C <= B and B <= A : return b if B <= A and A <= C : return a if C <= A and A <= B : return a if B <= C and C <= A : return d if A <= C and C <= B : return d # The main function that implements Introsort # low --> Starting index, # high --> Ending index # depthLimit --> recursion level def IntrosortUtil ( begin , end , depthLimit ): global arr size = end - begin if size < 16 : # if the data set is small, call insertion sort InsertionSort ( begin , end ) return if depthLimit
```

== 0 : # if the recursion limit is occurred call heap sort heapsort () return pivot = MedianOfThree (begin , begin + size // 2 , end) (arr [pivot], arr [end]) = (arr [end], arr [pivot]) # partitionPoint is partitioning index, # arr[partitionPoint] is now at right place partitionPoint = Partition (begin , end) # Separately sort elements before partition and after partition IntrosortUtil (begin , partitionPoint - 1 , depthLimit - 1) IntrosortUtil (partitionPoint + 1 , end , depthLimit - 1) # A utility function to begin the Introsort module def Introsort (begin , end): # initialise the depthLimit as 2 * log(length(data)) depthLimit = 2 * math . floor (math . log2 (end - begin)) IntrosortUtil (begin , end , depthLimit) # A utility function to print the array data def printArr (): print ('Arr: ' , arr) def main (): global arr arr = arr + [2 , 10 , 24 , 2 , 10 , 11 , 27 , 4 , 2 , 4 , 28 , 16 , 9 , 8 , 28 , 10 , 13 , 24 , 22 , 28 , 0 , 13 , 27 , 13 , 3 , 23 , 18 , 22 , 8 , 8] n = len (arr) Introsort (0 , n - 1) printArr () if __name__ == '__main__' : main () Output 0 2 2 2 3 4 4 8 8 8 9 10 10 10 11 13 13 13 16 18 22 22 23 24 24 27 27 28 28 28 Comment Article Tags: Article Tags: DSA