GeeksforGeeks
A computer science portal for geeks

IntroSort or Introspective sort

GeeksforGeeks
A computer science portal for geeks
Custom Search

Introsort(Introspective sort) is a comparison based sort that consists of three sorting phases. They are Quicksort, Heapsort, and Inse d 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.

Hire with us!

1. 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.

2. 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).

3. 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.

4. 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 requiremen 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.

1. If the number of elements in the input gets fewer, the Introsort performs Insertion sort for the input.
2. 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.
   1. 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
   2. 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).

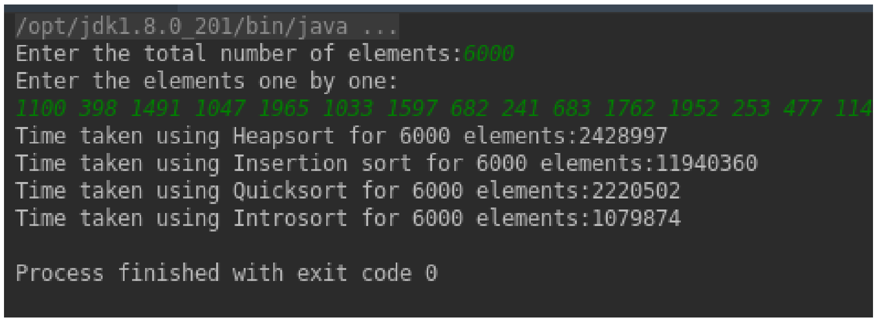
Why is Mergesort not used?
As the arrays are being dealt with the in-place concept where Quicksort outperforms Mergesort, we are not using Mergesort.

Can Introsort be applied everywhere?

1. If the data won't fit in an array, Introsort cannot be used.
2. 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).



```
/opt/jdk1.8.0_201/bin/java ...
Enter the total number of elements:6000
Enter the elements one by one:
1100 398 1491 1047 1965 1033 1597 682 241 683 1762 1952 253 477 114
Time taken using Heapsort for 6000 elements:2428997
Time taken using Insertion sort for 6000 elements:11940360
Time taken using Quicksort for 6000 elements:2220502
Time taken using Introsort for 6000 elements:1079874

Process finished with exit code 0
```

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)

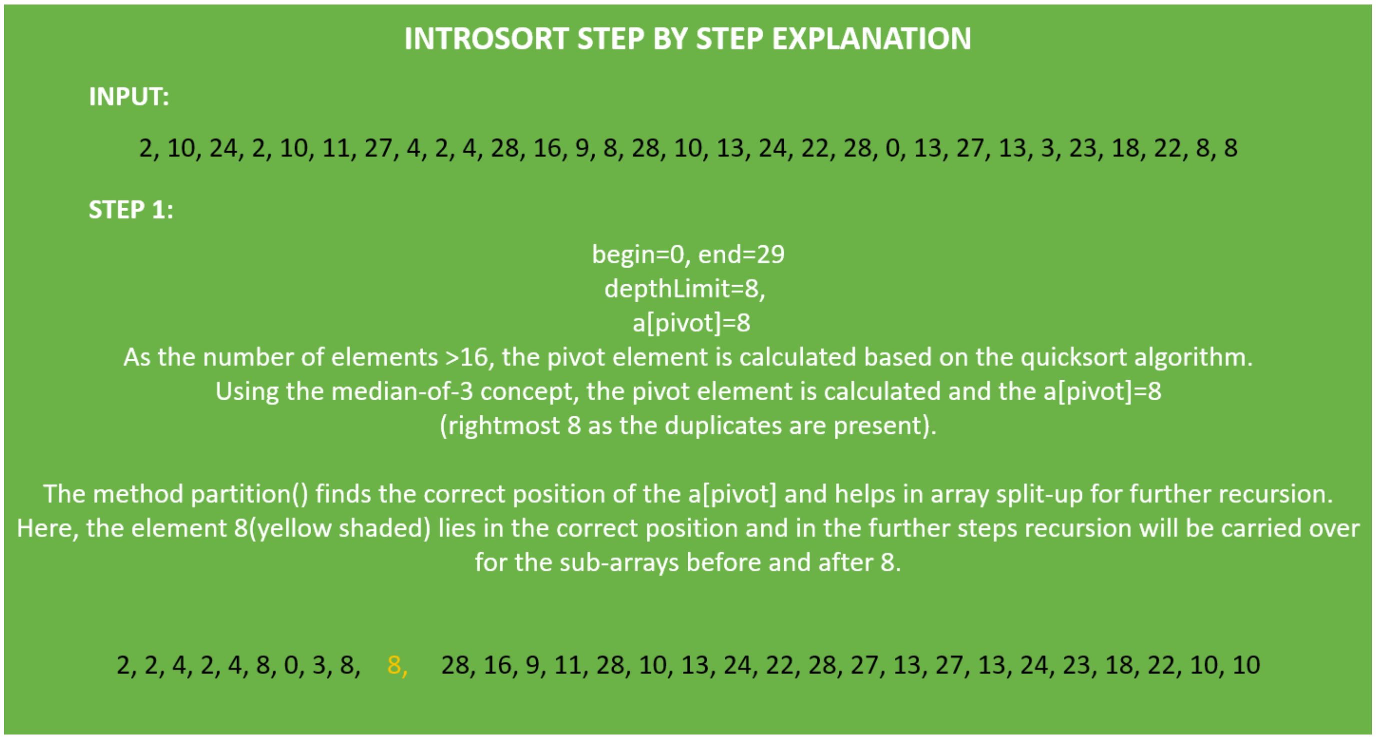
Average-case performance: O(nlogn)

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:

Step 1


INTROSORT STEP BY STEP EXPLANATION

INPUT:

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

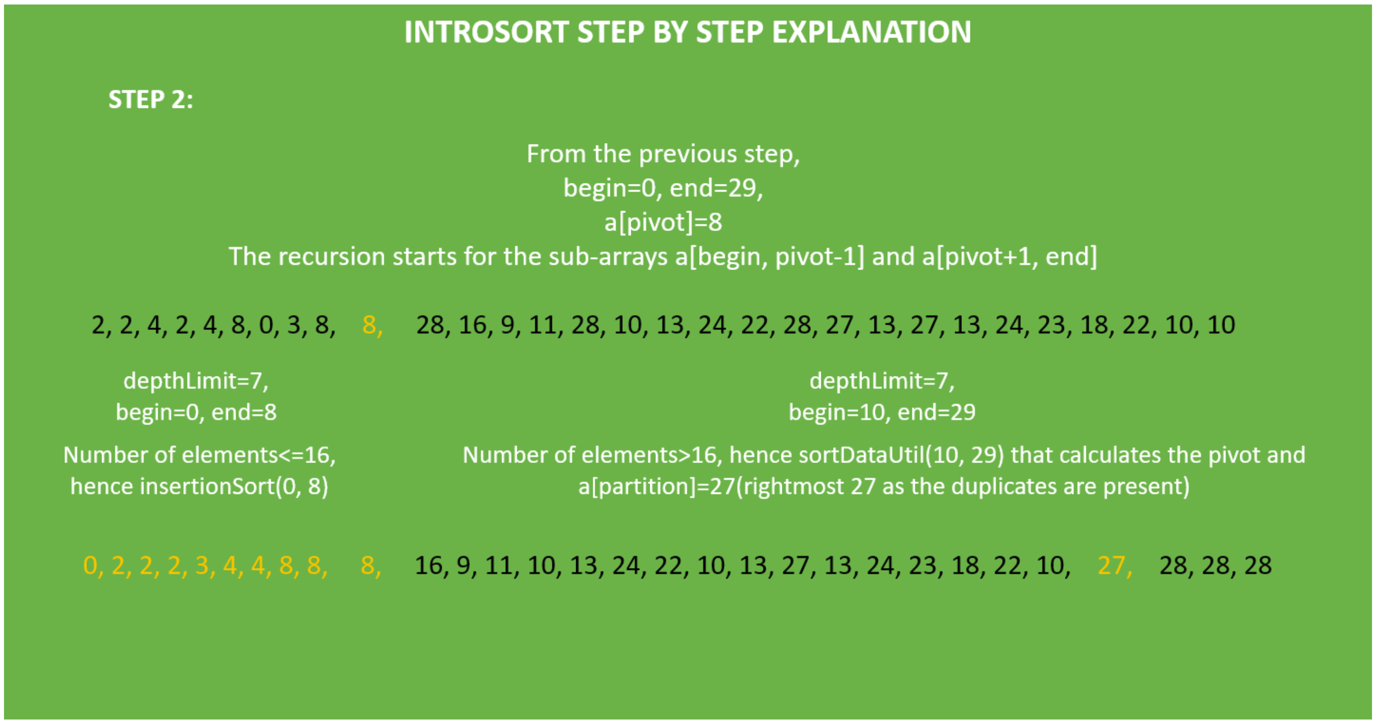
STEP 1:

begin=0, end=29
depthLimit=8,
a[pivot]=8
As the number of elements >16, the pivot element is calculated based on the quicksort algorithm.
Using the median-of-3 concept, the pivot element is calculated and the a[pivot]=8
(rightmost 8 as the duplicates are present).

The method partition() finds the correct position of the a[pivot] and helps in array split-up for further recursion.
Here, the element 8(yellow shaded) lies in the correct position and in the further steps recursion will be carried over for the sub-arrays before and after 8.

2, 2, 4, 2, 4, 8, 0, 3, 8, 8, 28, 16, 9, 11, 28, 10, 13, 24, 22, 28, 27, 13, 27, 13, 24, 23, 18, 22, 10, 10

Step 2


INTROSORT STEP BY STEP EXPLANATION

STEP 2:

From the previous step,
begin=0, end=29,
a[pivot]=8
The recursion starts for the sub-arrays a[begin, pivot-1] and a[pivot+1, end]

2, 2, 4, 2, 4, 8, 0, 3, 8, 8, 28, 16, 9, 11, 28, 10, 13, 24, 22, 28, 27, 13, 27, 13, 24, 23, 18, 22, 10, 10

depthLimit=7,
begin=0, end=8

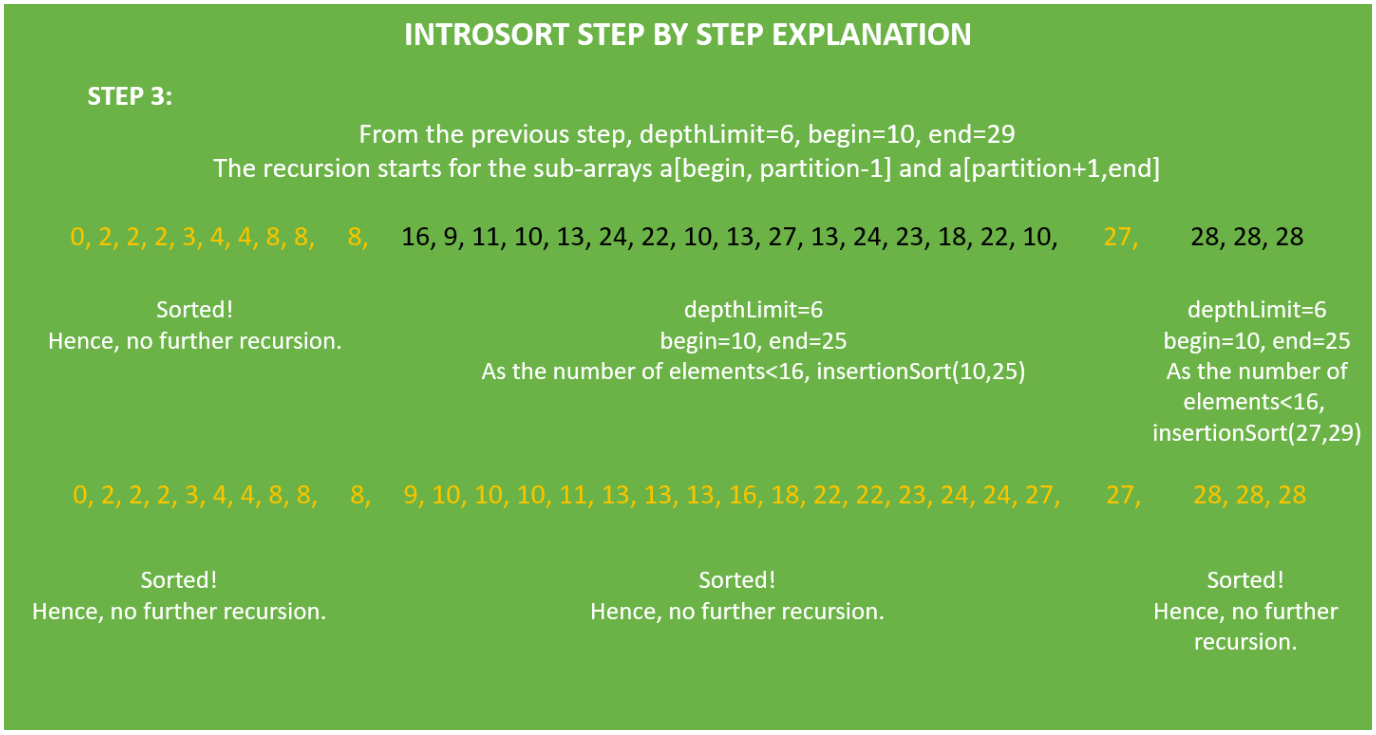
Number of elements<=16,
hence insertionSort(0, 8)

depthLimit=7,
begin=10, end=29

Number of elements>16, hence sortDataUtil(10, 29) that calculates the pivot and a[partition]=27(rightmost 27 as the duplicates are present)

0, 2, 2, 2, 3, 4, 4, 8, 8, 8, 16, 9, 11, 10, 13, 24, 22, 10, 13, 27, 13, 24, 23, 18, 22, 10, 27, 28, 28, 28

Step 3



INTROSORT STEP BY STEP EXPLANATION

STEP 3:

From the previous step, depthLimit=6, begin=10, end=29
The recursion starts for the sub-arrays a[begin, partition-1] and a[partition+1,end]

0, 2, 2, 2, 3, 4, 4, 8, 8, 8, 16, 9, 11, 10, 13, 24, 22, 10, 13, 27, 13, 24, 23, 18, 22, 10, 27, 28, 28, 28

Sorted!
Hence, no further recursion.

depthLimit=6
begin=10, end=25
As the number of elements<16, insertionSort(10,25)

depthLimit=6
begin=10, end=25
As the number of elements<16, insertionSort(27,29)

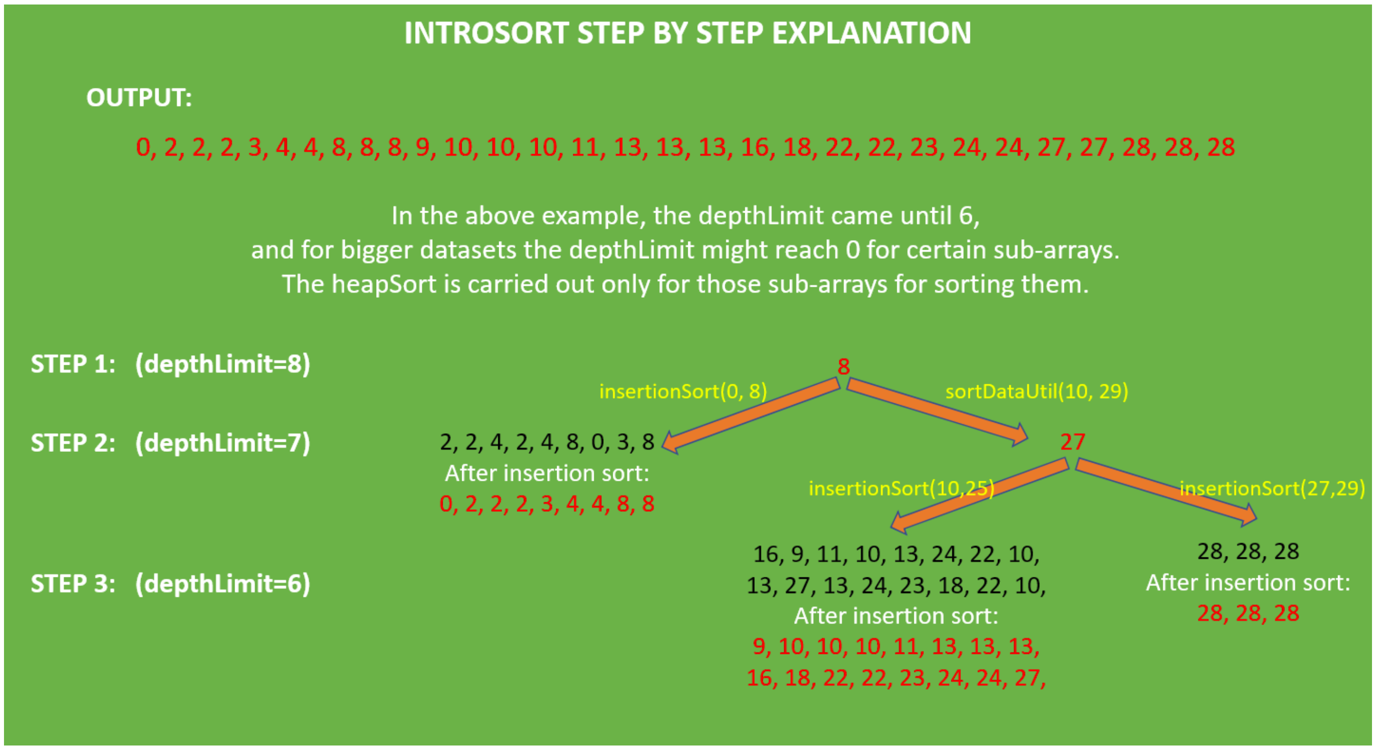
0, 2, 2, 2, 3, 4, 4, 8, 8, 8, 9, 10, 10, 10, 11, 13, 13, 13, 16, 18, 22, 22, 23, 24, 24, 27, 27, 28, 28, 28

Sorted!
Hence, no further recursion.

Sorted!
Hence, no further recursion.

Sorted!
Hence, no further recursion.

Step 4



INTROSORT STEP BY STEP EXPLANATION

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

In the above example, the depthLimit came until 6,
and for bigger datasets the depthLimit might reach 0 for certain sub-arrays.
The heapSort is carried out only for those sub-arrays for sorting them.

STEP 1: (depthLimit=8)

STEP 2: (depthLimit=7)

STEP 3: (depthLimit=6)

8
insertionSort(0, 8)
sortDataUtil(10, 29)
2, 2, 4, 2, 4, 8, 0, 3, 8
After insertion sort:
0, 2, 2, 2, 3, 4, 4, 8, 8
27
insertionSort(10,25)
insertionSort(27,29)
16, 9, 11, 10, 13, 24, 22, 10, 13, 27, 13, 24, 23, 18, 22, 10,
After insertion sort:
9, 10, 10, 10, 11, 13, 13, 13, 16, 18, 22, 22, 23, 24, 24, 27,
28, 28, 28
After insertion sort:
28, 28, 28

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
```

Below is the implementation of the above approach:

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();
    }
}
```

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.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
```

Recommended Posts:

Know Your Sorting Algorithm | Set 2 (Introsort- C++'s Sorting Weapon)
Comparison among Bubble Sort, Selection Sort and Insertion Sort
Why Quick Sort preferred for Arrays and Merge Sort for Linked Lists?
Program to sort an array of strings using Selection Sort
Insertion sort to sort even and odd positioned elements in different orders
Bucket Sort To Sort an Array with Negative Numbers
Odd Even Transposition Sort / Brick Sort using pthreads
Java Program for Odd-Even Sort / Brick Sort
Serial Sort v/s Parallel Sort in Java
Quick Sort vs Merge Sort
C/C++ Program for Odd-Even Sort / Brick Sort
Odd-Even Sort / Brick Sort
Sort all even numbers in ascending order and then sort all odd numbers in descending order
How to sort an Array in C# | Array.Sort() Method Set – 2
How to sort a list in C# | List.Sort() Method Set -1

Lokesh Karthikeyan
Check out this Author's contributed articles.

If you like GeeksforGeeks and would like to contribute, you can also write an article using contribute.geeksforgeeks.org or mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please Improve this article if you find anything incorrect by clicking on the "Improve Article" button below.

Improved By : Lokesh Karthikeyan

Article Tags : Algorithms Divide and Conquer Sorting Heap Sort Merge Sort Quick Sort

Practice Tags : Divide and Conquer Sorting Merge Sort Algorithms

2

0

☐ To-do ☐ Done

No votes yet.

Feedback/ Suggest Improvement Add Notes Improve Article

Please write to us at contribute@geeksforgeeks.org to report any issue with the above content.

Writing code in comment? Please use ide.geeksforgeeks.org, generate link and share the link here.

Load Comments

A computer science portal for geeks

5th Floor, A-118,
Sector-136, Noida, Uttar Pradesh - 201305
feedback@geeksforgeeks.org

COMPANY
About Us
Careers
Privacy Policy
Contact Us

LEARN
Algorithms
Data Structures
Languages
CS Subjects
Video Tutorials

PRACTICE
Courses
Company-wise
Topic-wise
How to begin?

CONTRIBUTE
Write an Article
Write Interview Experience
Internships
Videos

@geeksforgeeks, Some rights reserved

▲