**Hybrid sorting algorithm**: I have used a hybrid sorting algorithm which name is introsort algorithm. Simply putting, it is the best sorting algorithm around. It is a hybrid sorting algorithm, which means that it uses more than one sorting algorithms as a routine. Which, Best Case- O (N log N),  
 Average Case- O (N log N),  
 Worse Case- O (N log N).

**Used parameters**: Introsort being a hybrid sorting algorithm uses three sorting algorithm to minimize the running time, [Quicksort](http://quiz.geeksforgeeks.org/quick-sort/), [Heapsort](http://quiz.geeksforgeeks.org/heap-sort/) and [Insertion Sort](http://quiz.geeksforgeeks.org/insertion-sort/).

>>It is a known and established fact that insertion sort is the most optimal comparison-based sorting algorithm for small arrays. It has a good locality of reference. It is an adaptive sorting algorithm, i.e- it outperforms all the other algorithms if the array elements are partially sorted.

>>This is solely because of memory requirements. Merge sort requires O(N) space whereas Heapsort is an in-place O(1) space algorithm.

>>This question is same as why Quicksort generally outperforms Heapsort.The answer is, although Heapsort also being O(N log N) in average as well as worse case and O(1) space also, we still don’t use it when the partition size is under the limit because the extra hidden constant factor in Heapsort is quite larger than that of Quicksort.

I think it will be better one.

**Pseudocode**: A Program to sort the array using Introsort.

A utility function to swap the values pointed by the two pointers

swapValue(\*a,\*b)

    \*temp <- a

    a <- b

    b <- temp

    return

Function to sort an array using insertion sort

InsertionSort(arr[],\*begin, \*end)

    Get the left and the right index of the subarray to be sorted

    left <- begin - arr

    right <- end - arr

    for (i = left+1; i <= right; i++)

        key <- arr[i]

        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

\*Partition(int arr[], int low, int high)

    pivot <- arr[high];    pivot

i <- (low - 1)

for (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

\*MedianOfThree(\* a, \* b, \* 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 <= \*c)

        return (b)

A Utility function to perform intro sort

IntrosortUtil(arr[],\* begin,\* end, depthLimit)

Count the number of elements

     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

     \* 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

Introsort(int arr[], int \*begin, int \*end)

    depthLimit = 2 \* log(end-begin)

    Perform a recursive Introsort

    IntrosortUtil(arr, begin, end, depthLimit)

 return

A utility function to print an array of size n

printArray(int arr[], int n)

   for ( i=0; i < n; i++)

       Print arr

Driver program to test Introsort

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)

**Work process simulation**: Introsort begins with quicksort and if the recursion depth goes more than a particular limit it switches to Heapsort to avoid Quicksort’s worse case O(N2) time complexity. So first it creates a partition. Three cases arises from here.

1. If the partition size is such that there is a possibility to exceed the maximum depth limit then the Introsort switches to Heapsort. We define the maximum depth limit as 2\*log(N)
2. If the partition size is too small then Quicksort decays to Insertion Sort. We define this cutoff as 16 (due to research). So if the partition size is less than 16 then we will do insertion sort.
3. If the partition size if under the limit and not too small (i.e- between 16 and 2\*log(N)), then it performs a simple quicksort.

#If array size is < 16 then I have done insertion sort.

Suppose the input data is: 12,11,13,5,6.

**Step1>> 12**, 11, 13, 5, 6

**step2>> 11, 12**, 13, 5, 6  
**step3>> 11, 12, 13**, 5, 6  
**step4>> 5, 11, 12, 13**, 6  
**step5>> 5, 6, 11, 12, 13**

#If the depth is zero then heapsort

Suppose the input data is: 4,10,3,5,1.

Step1>>

4(0)

/ \

10(1) 3(2)

/ \

5(3) 1(4)

Step2>>

4(0)

/ \

10(1) 3(2)

/ \

5(3) 1(4)

Step3>>

10(0)

/ \

5(1) 3(2)

/ \

4(3) 1(4)

#Else use a median-of-three concept to find a good pivot Swap the values pointed by the two pointers Perform Quick Sort.

Suppose the array = {10, 80, 30, 90, 40, 50, 70}

Indexes: 0 1 2 3 4 5 6

Step1>>

Low = 0, high = 6, pivot = arr[h] = 70

Initialize index of smaller element, **i = -1**

Traverse elements from j = low to high-1

**j = 0** : Since arr[j] <= pivot, do i++ and swap(arr[i], arr[j])

**i = 0**

arr [] = {10, 80, 30, 90, 40, 50, 70}

Step2>>

**j = 1**: Since arr[j] > pivot, do nothing

**j = 2**: Since arr[j] <= pivot, do i++ and swap(arr[i], arr[j])

**i = 1**

arr [] = {10, 30, 80, 90, 40, 50, 70}

Step3>>

**j = 3**: Since arr[j] > pivot, do nothing

**j = 4**: Since arr[j] <= pivot, do i++ and swap(arr[i], arr[j])

**i = 2**

arr [] = {10, 30, 40, 90, 80, 50, 70}

Step4>>

**j = 5**: Since arr[j] <= pivot, do i++ and swap arr[i] with arr[j]

**i = 3**

arr [] = {10, 30, 40, 50, 80, 90, 70}

Step5>>

We come out of loop because j is now equal to high-1.

arr[] = {10, 30, 40, 50, 70, 90, 80}

Now 70 is at its correct place. All elements smaller than

70 are before it and all elements greater than 70 are after it.

arr [] = {10, 30, 40, 50, 70, 80, 90}

**Source code**: Introsort source code has given below.

#include<bits/stdc++.h>

using namespace std;

void swapValue(int \*a, int \*b){

int \*temp = a;

a = b;

b = temp;

return;

}

void InsertionSort(int arr[], int \*begin, int \*end){

int left = begin - arr;

int right = end - arr;

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 = j-1;

}

arr[j+1] = key;

}

return;

}

int\* Partition(int arr[], int low, int high){

int pivot = arr[high];

int i = (low - 1);

for (int j = low; j <= high- 1; j++){

if (arr[j] <= pivot){

i++;

swap(arr[i], arr[j]);

}

}

swap(arr[i + 1], arr[high]);

return (arr + i + 1);

}

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 <= \*c)

return (b);

}

void IntrosortUtil(int arr[], int \* begin,int \* end, int depthLimit){

int size = end - begin;

if (size < 16){

InsertionSort(arr, begin, end);

return;

}

if (depthLimit == 0){

make\_heap(begin, end+1);

sort\_heap(begin, end+1);

return;

}

int \* pivot = MedianOfThree(begin, begin+size/2, end);

swapValue(pivot, end);

int \* partitionPoint = Partition(arr, begin-arr, end-arr);

IntrosortUtil(arr, begin, partitionPoint-1, depthLimit - 1);

IntrosortUtil(arr, partitionPoint + 1, end, depthLimit - 1);

return;

}

void Introsort(int arr[], int \*begin, int \*end){

int depthLimit = 2 \* log(end-begin);

IntrosortUtil(arr, begin, end, depthLimit);

return;

}

void printArray(int arr[], int n){

for (int i=0; i < n; i++)

cout<< arr[i] <<" ";

cout<<endl;

}

int main()

{

int n;

int arr[100];

cout<<"Enter the size of ur array: ";

cin>>n;

cout<<"Enter the data of ur array: ";

for(int i=0;i<n;i++){

cin>>arr[i];

}

Introsort(arr, arr, arr+n-1);

printArray(arr, n);

}

**Uses of applications**: We can use this sorting algorithm in three types of sector.

1. Where the array data is less than 16 size then it will execute insertion sort. It is a known and established fact that insertion sort is the most optimal comparison-based sorting algorithm for small arrays.

2. If the depth is zero then heapsort will execute. HeapSort is not used much in practice, but can be useful in real time (or time bound where QuickSort doesn’t fit) embedded systems where less space is available (MergeSort doesn’t fit)

3. Third one is if Quick Sort in its general form is an in-place sort (i.e. it doesn’t require any extra storage) whereas merge sort requires O(N) extra storage, N denoting the array size which may be quite expensive. Allocating and de-allocating the extra space used for merge sort increases the running time of the algorithm.

So we can see that if user input sort, long, narrow space pc data needed to sort then they can use this type of in their application.

That’s all about hybrid sort.

Thank you