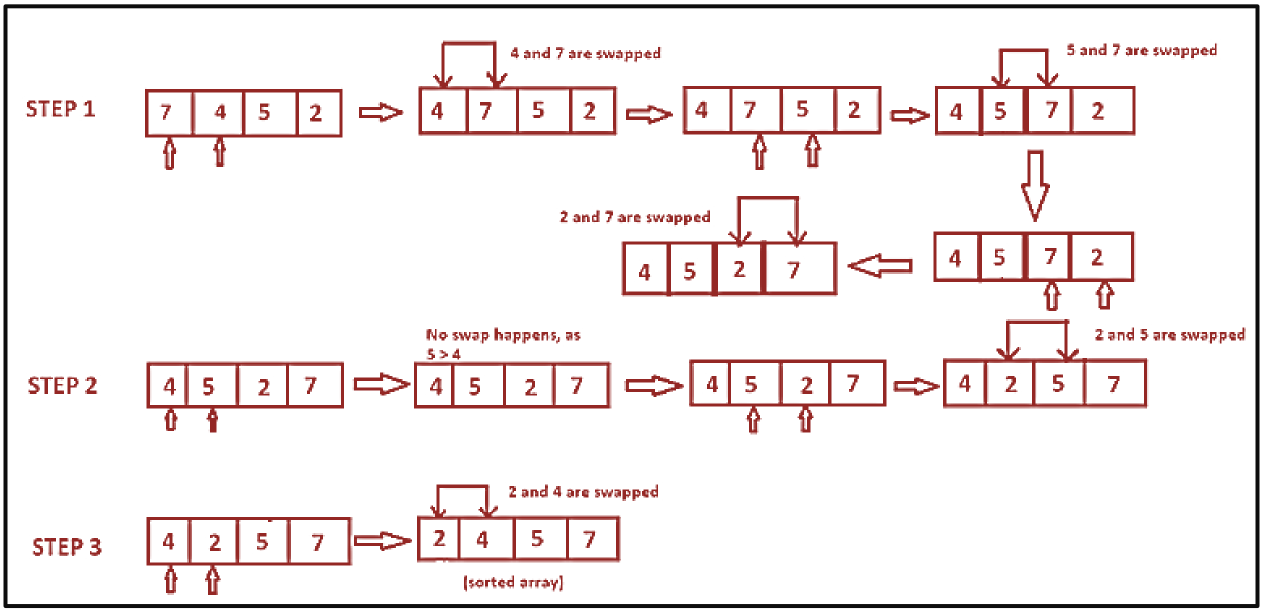
Assignment 3

Sorting Algorithms

Name : duaa alqawaqzeh
Email : duaa.alqawaqzeh@gmail.com

Bubble sort

Bubble Sort is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in wrong order.



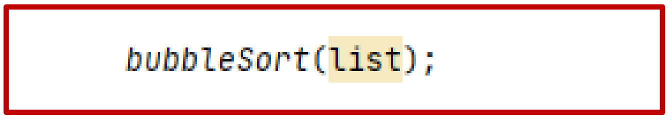
Worst and Average Case Time Complexity: O(n^2). Worst case occurs when array is reverse sorted.

Best Case Time Complexity: O(n). Best case occurs when array is already sorted.

Auxiliary Space: O(1) Selection Sort sorts *in-place*, meaning we do not need to allocate any memory for the sort to occur

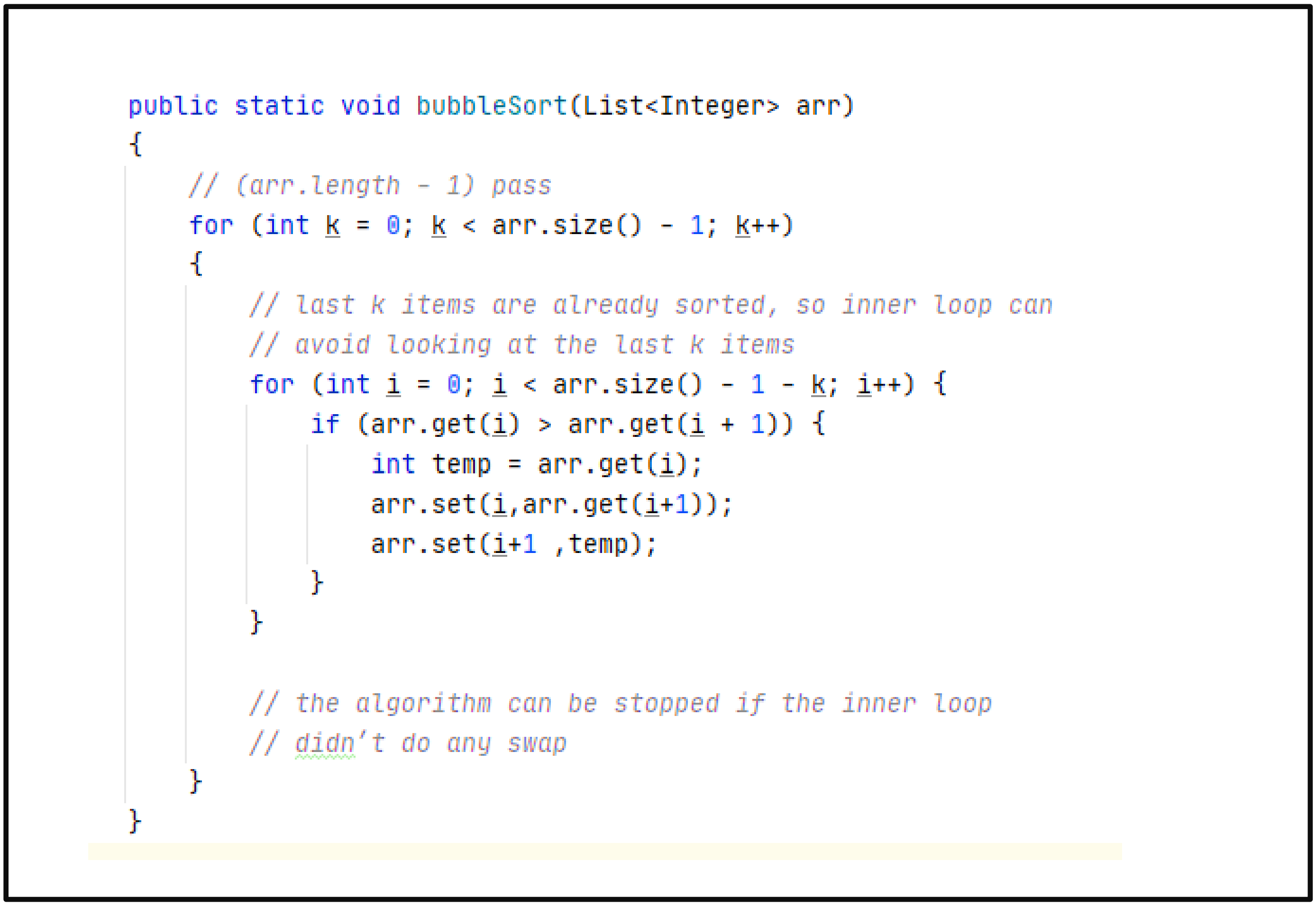
Implementation :

Inside sort(List list)method =>



```
bubbleSort(list);
```

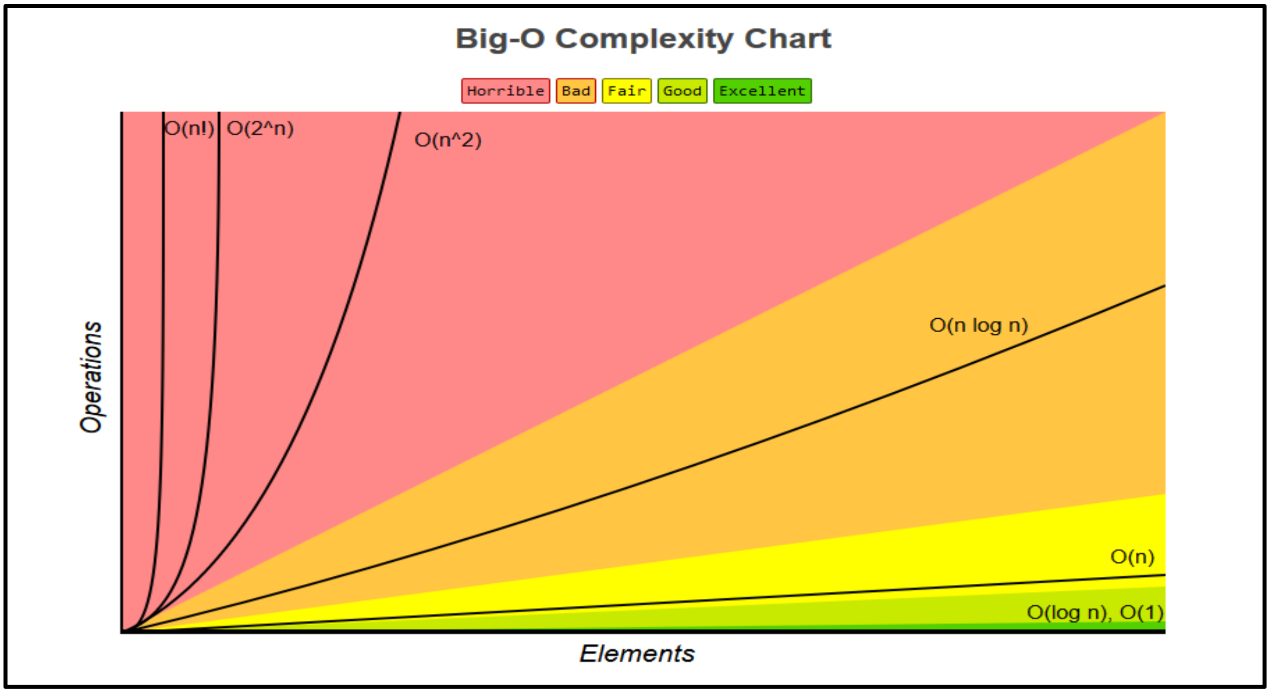
Functions :

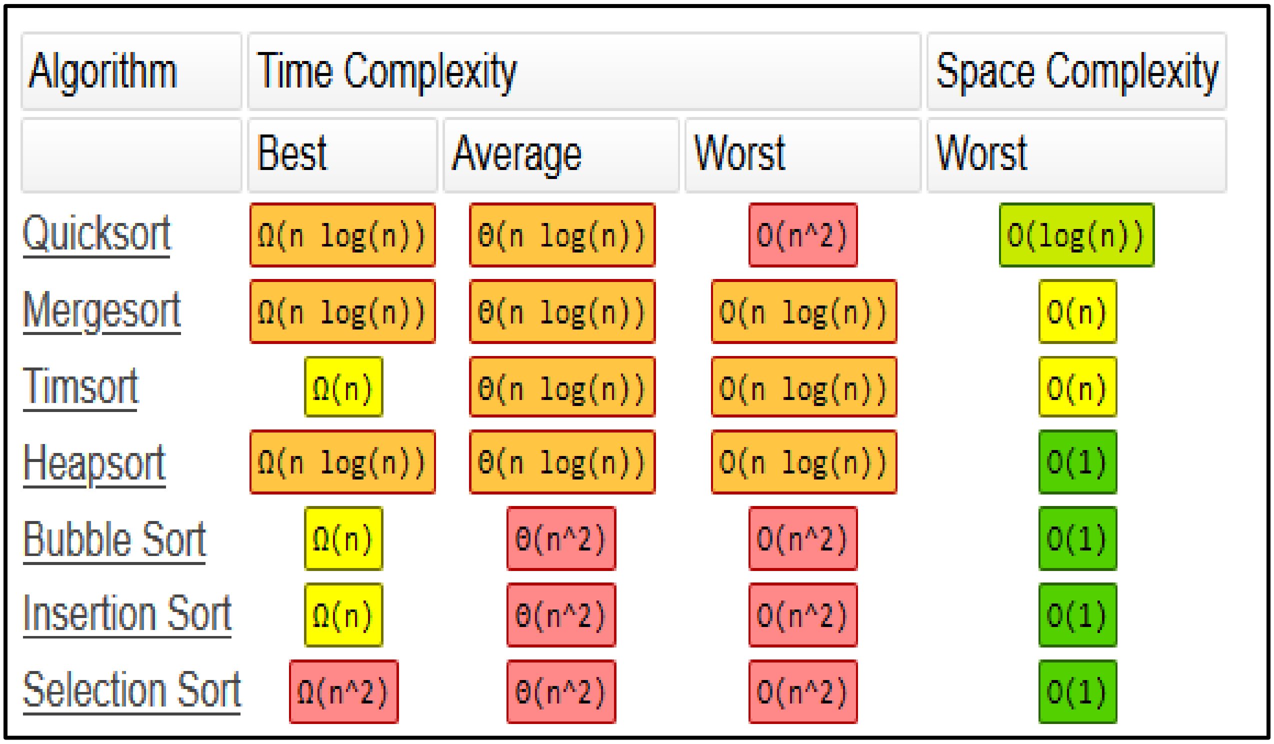


```
public static void bubbleSort(List<Integer> arr)
{
    // (arr.length - 1) pass
    for (int k = 0; k < arr.size() - 1; k++)
    {
        // last k items are already sorted, so inner loop can
        // avoid looking at the last k items
        for (int i = 0; i < arr.size() - 1 - k; i++) {
            if (arr.get(i) > arr.get(i + 1)) {
                int temp = arr.get(i);
                arr.set(i,arr.get(i+1));
                arr.set(i+1 ,temp);
            }
        }

        // the algorithm can be stopped if the inner loop
        // didn’t do any swap
    }
}
```

Results :

It took too long without results.





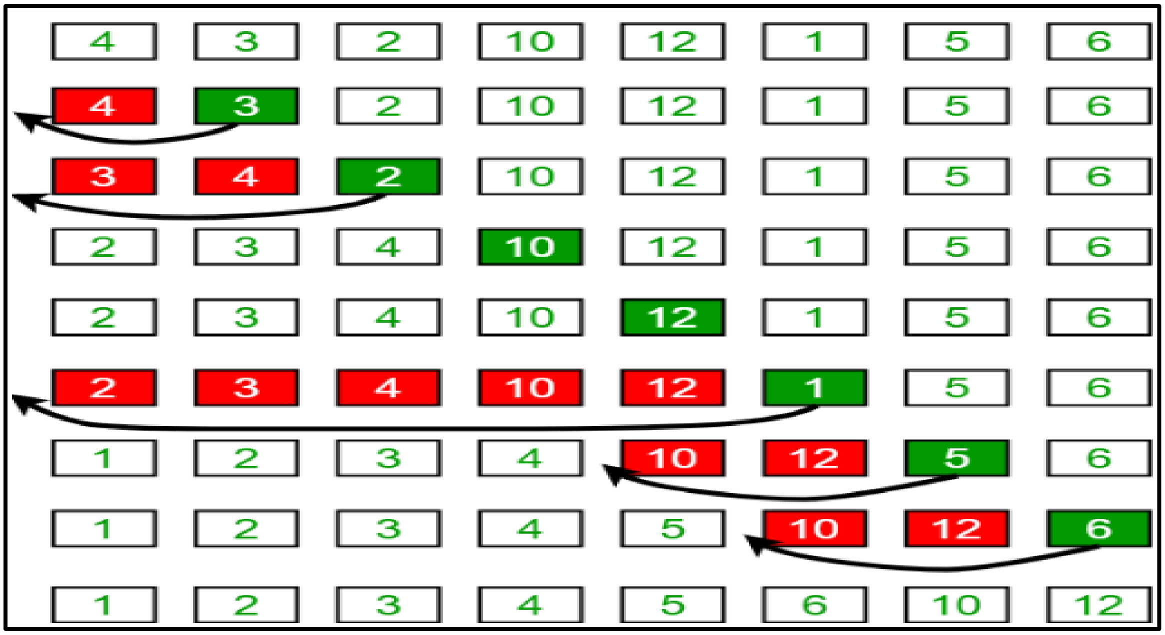
Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	Worst
Quicksort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(log(n))
Mergesort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(n)
Timsort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)
Heapsort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(1)
Bubble Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Insertion Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Selection Sort	Ω(n^2)	Θ(n^2)	O(n^2)	O(1)

Insertion sort

The array is virtually split into a sorted and an unsorted part. Values from the unsorted part are picked and placed at the correct position in the sorted part.

Insertion sort is used when number of elements is small. It can also be useful when input array is almost sorted, only few elements are misplaced in complete big array.

To sort an array of size n in ascending order:
1: Iterate from arr[1] to arr[n] over the array.
2: Compare the current element (key) to its predecessor.
3: If the key element is smaller than its predecessor, compare it to the elements before. Move the greater elements one position up to make space for the swapped element.



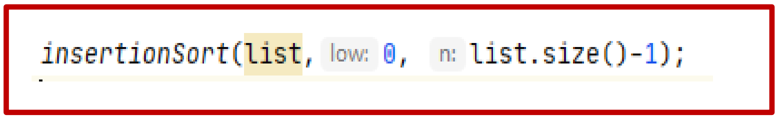
Worst and Average Case Time Complexity: O(n^2). The worst case for insertion sort will occur when the input list is in decreasing order.

Best Case Time Complexity: O(n). If the input array is already in sorted order.

Auxiliary Space: O(1)Selection Sort sorts *in-place*, meaning we do not need to allocate any memory for the sort to occur

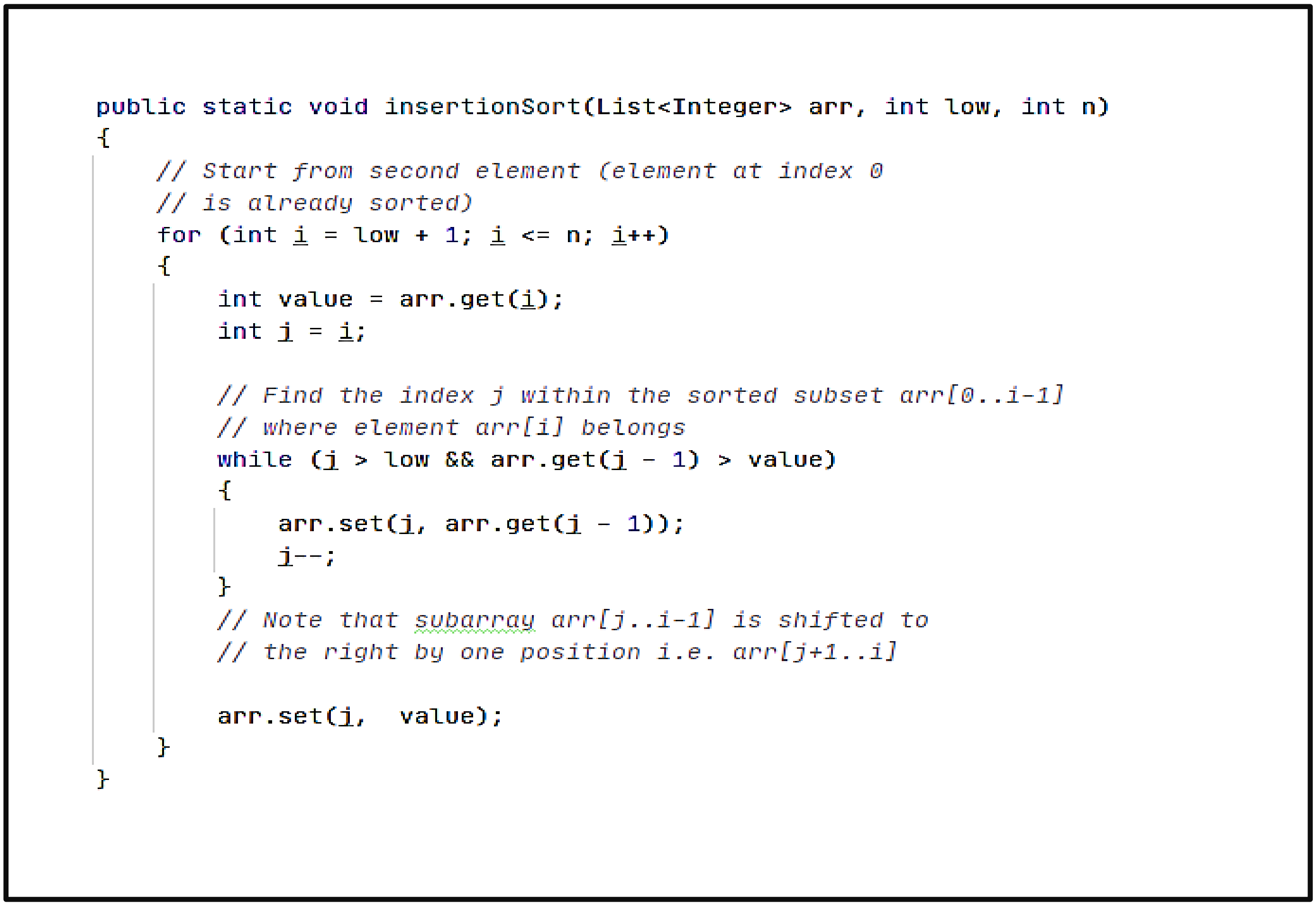
Implementation :

Inside sort(List list)method =>



```
insertionSort(list, low: 0, n: list.size()-1);
```

Functions :



```
public static void insertionSort(List<Integer> arr, int low, int n)
{
    // Start from second element (element at index 0
    // is already sorted)
    for (int i = low + 1; i <= n; i++)
    {
        int value = arr.get(i);
        int j = i;

        // Find the index j within the sorted subset arr[0..i-1]
        // where element arr[i] belongs
        while (j > low && arr.get(j - 1) > value)
        {
            arr.set(j, arr.get(j - 1));
            j--;
        }
        // Note that subarray arr[j..i-1] is shifted to
        // the right by one position i.e. arr[j+1..i]

        arr.set(j,  value);
    }
}
```

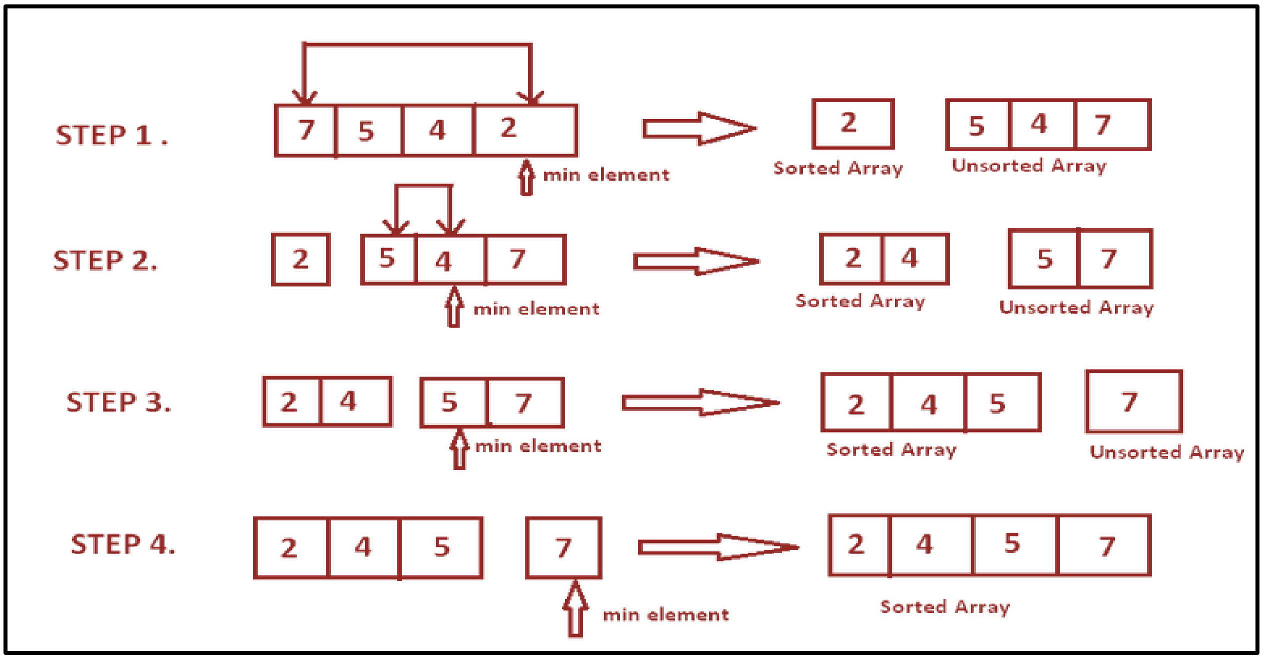
Results :

It took too long without results.

Selection sort

Selection Sort also divides the array into a sorted and unsorted subarray. Though, this time, the sorted subarray is formed by inserting the minimum element of the unsorted subarray at the end of the sorted array, by swapping:

1) The subarray which is already sorted.
2) Remaining subarray which is unsorted.



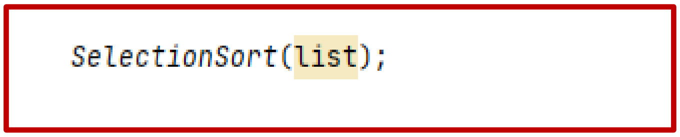
Worst and Average Case Time Complexity: O(n^2). In worst case,each element is compared with all the other elements in the sorted array. For elements, there will be comparisons. Therefore, the time complexity is O(n^2).

Best Case Time Complexity: O(n^2)In the best case, we already have a sorted array but we need to go through the array **O(n^2).**

Auxiliary Space: O(1)Selection Sort sorts *in-place*, meaning we do not need to allocate any memory for the sort to occur

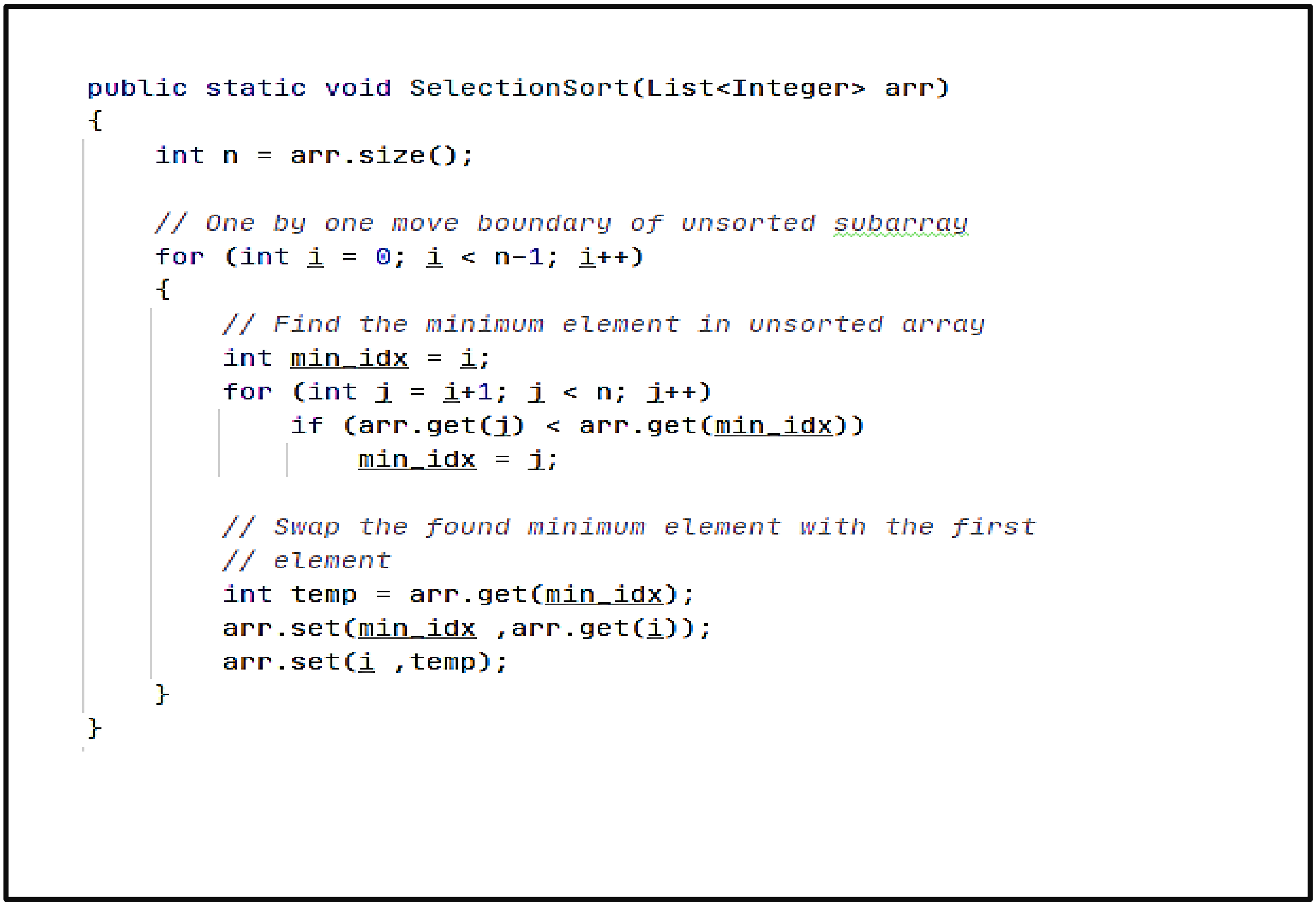
Implementation :

Inside sort(List list)method =>



```
SelectionSort(list);
```

Functions :



```
public static void SelectionSort(List<Integer> arr)
{
    int n = arr.size();

    // One by one move boundary of unsorted subarray
    for (int i = 0; i < n-1; i++)
    {
        // Find the minimum element in unsorted array
        int min_idx = i;
        for (int j = i+1; j < n; j++)
            if (arr.get(j) < arr.get(min_idx))
                min_idx = j;

        // Swap the found minimum element with the first
        // element
        int temp = arr.get(min_idx);
        arr.set(min_idx ,arr.get(i));
        arr.set(i ,temp);
    }
}
```

Results :

It took too long without results.

HeapSort

Heap sort is a comparison based sorting technique based on Binary Heap data structure.

A Binary Heap is a Complete Binary Tree where items are stored in a special order such that value in a parent node is greater(or smaller) than the values in its two children nodes .

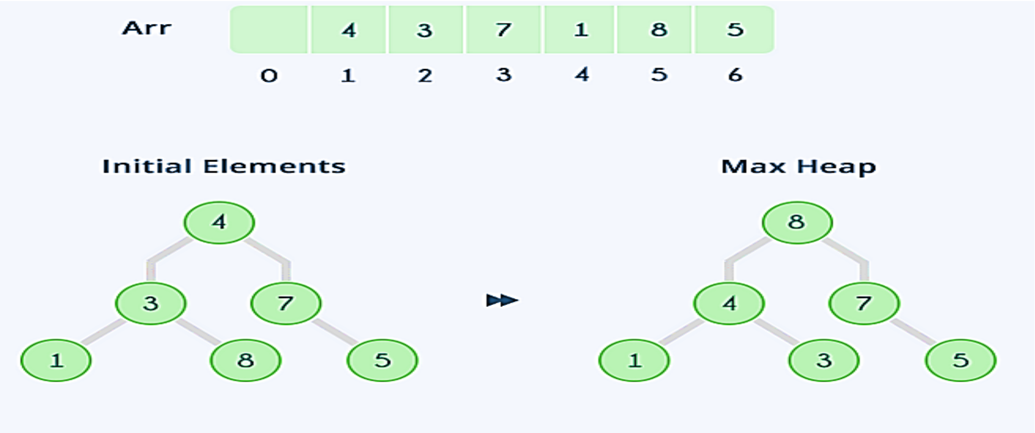
Heap Sort Algorithm for sorting in increasing order:

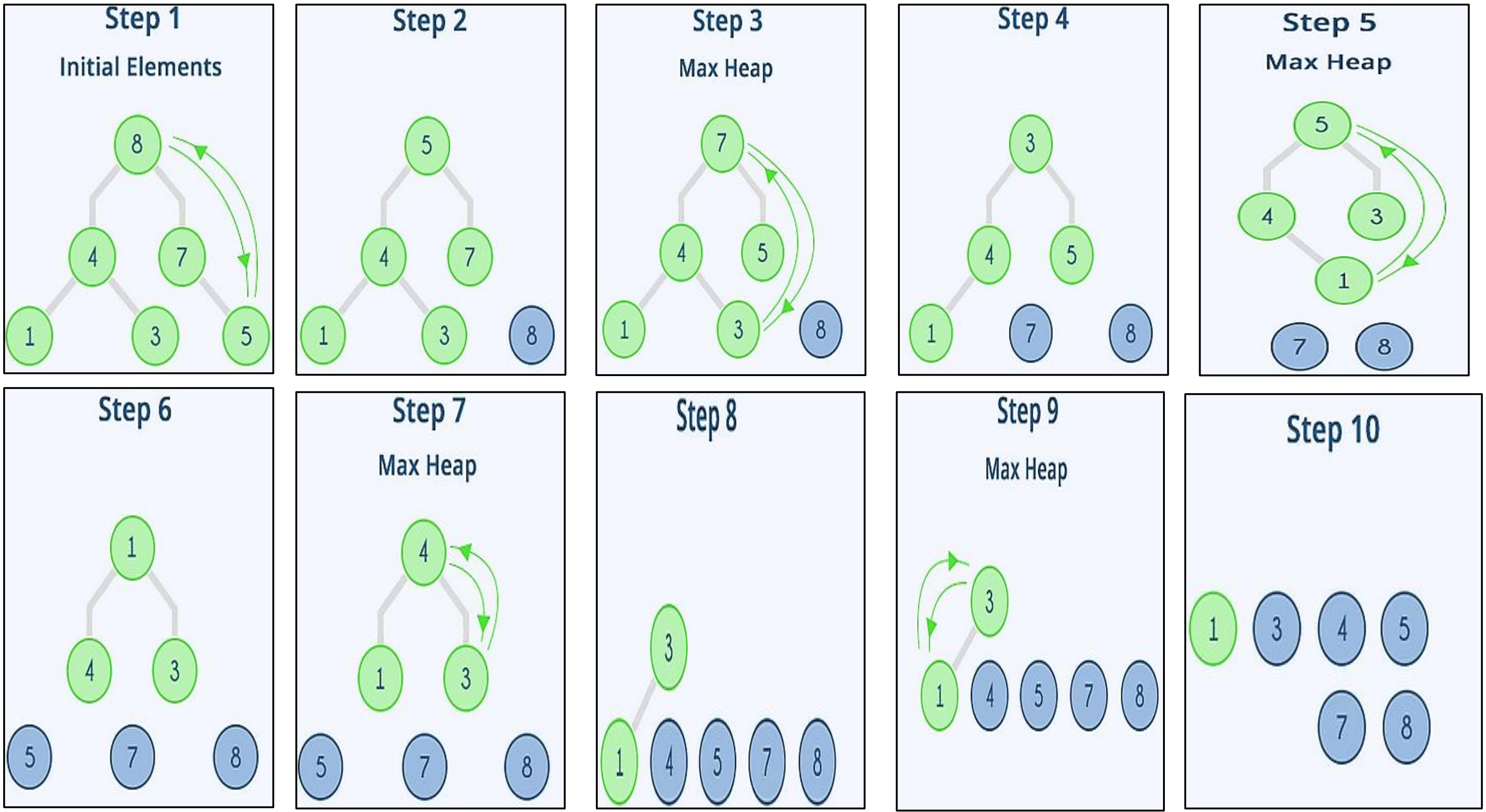
1. Build a max heap from the input data.
2. At this point, the largest item is stored at the root of the heap. Replace it with the last item of the heap followed by reducing the size of heap by 1. Finally, heapify the root of the tree.
3. Repeat step 2 while size of heap is greater than 1

Heapify procedure can be applied to a node only if its children nodes are heapified. So the heapification must be performed in the bottom-up order.

Worst and Average Case Time Complexity and best Case: O(nlogn).

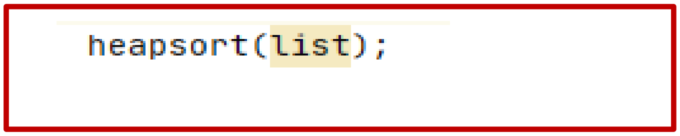
Time complexity of heapify is O(Logn). Time complexity of createAndBuildHeap() is O(n) and overall time complexity of Heap Sort is O(nLogn).





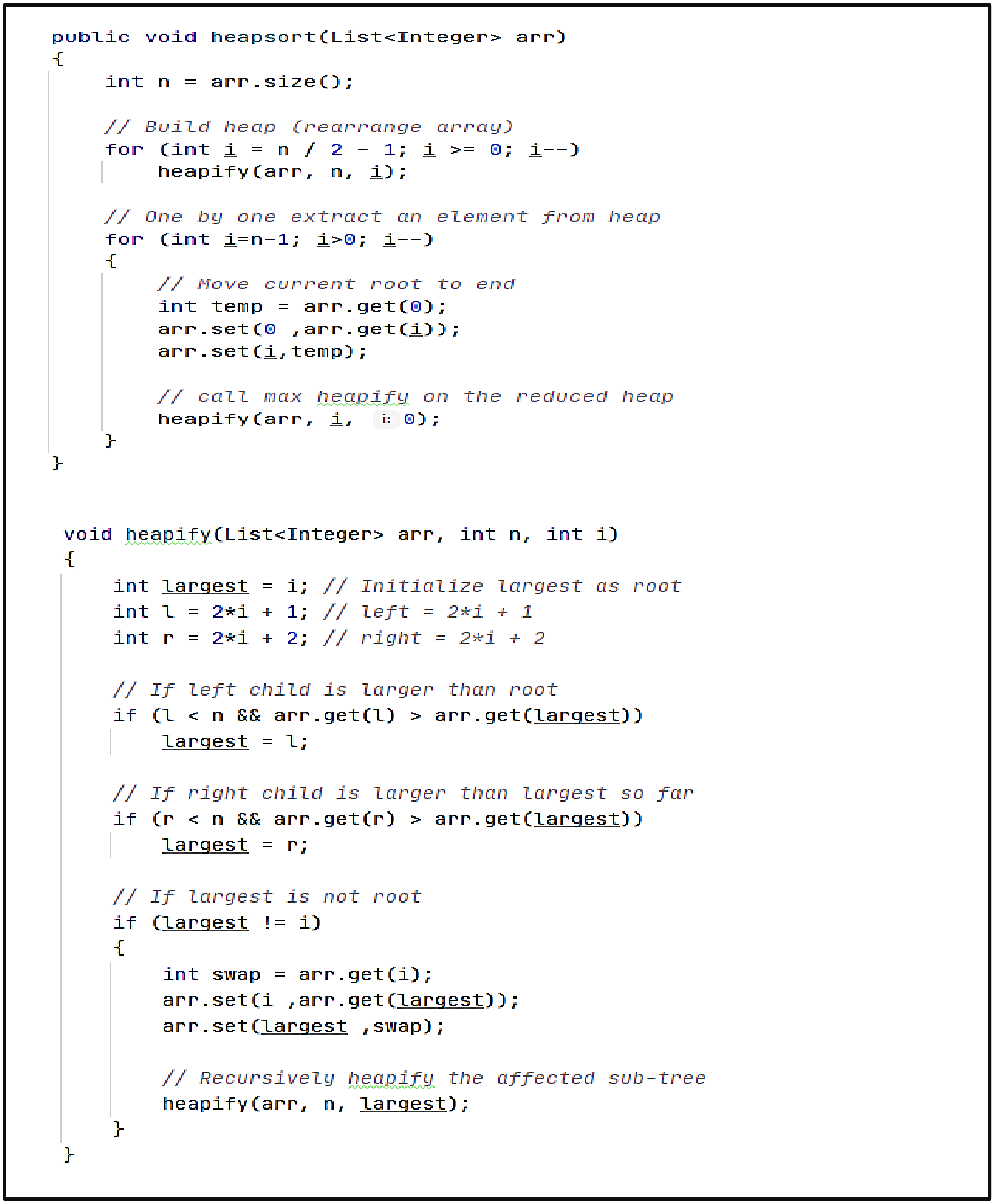
Implementation :

Inside sort(List list)method =>



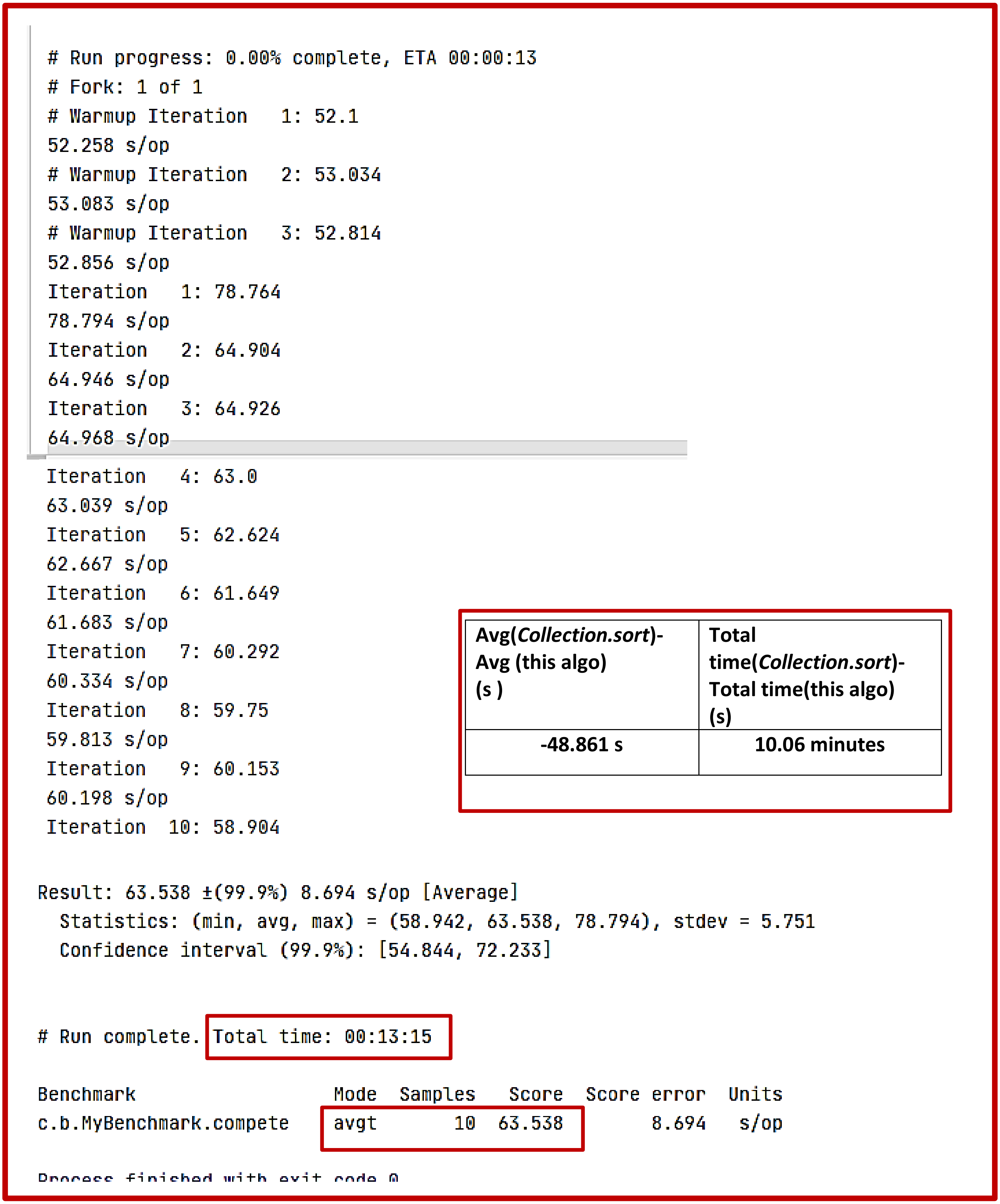
```
heapsort(list);
```

Functions :



```
public void heapsort(List<Integer> arr)
{
    int n = arr.size();

    // Build heap (rearrange array)
    for (int i = n / 2 - 1; i >= 0; i--)
        heapify(arr, n, i);

    // One by one extract an element from heap
    for (int i=n-1; i>0; i--)
    {
        // Move current root to end
        int temp = arr.get(0);
        arr.set(0 ,arr.get(i));
        arr.set(i,temp);

        // call max heapify on the reduced heap
        heapify(arr, i,  i: 0);
    }
}


 void heapify(List<Integer> arr, int n, int i)
 {
     int largest = i; // Initialize largest as root
     int l = 2*i + 1; // left = 2*i + 1
     int r = 2*i + 2; // right = 2*i + 2

     // If left child is larger than root
     if (l < n && arr.get(l) > arr.get(largest))
         largest = l;

     // If right child is larger than largest so far
     if (r < n && arr.get(r) > arr.get(largest))
         largest = r;

     // If largest is not root
     if (largest != i)
     {
         int swap = arr.get(i);
         arr.set(i ,arr.get(largest));
         arr.set(largest ,swap);

         // Recursively heapify the affected sub-tree
         heapify(arr, n, largest);
     }
 }
```

Results :



```
# Run progress: 0.00% complete, ETA 00:00:13
# Fork: 1 of 1
# Warmup Iteration   1: 52.1
52.258 s/op
# Warmup Iteration   2: 53.034
53.083 s/op
# Warmup Iteration   3: 52.814
52.856 s/op
Iteration   1: 78.764
78.794 s/op
Iteration   2: 64.904
64.946 s/op
Iteration   3: 64.926
64.968 s/op
Iteration   4: 63.0
63.039 s/op
Iteration   5: 62.624
62.667 s/op
Iteration   6: 61.649
61.683 s/op
Iteration   7: 60.292
60.334 s/op
Iteration   8: 59.75
59.813 s/op
Iteration   9: 60.153
60.198 s/op
Iteration  10: 58.904


Result: 63.538 ±(99.9%) 8.694 s/op [Average]
  Statistics: (min, avg, max) = (58.942, 63.538, 78.794), stdev = 5.751
  Confidence interval (99.9%): [54.844, 72.233]


# Run complete. Total time: 00:13:15

Benchmark                    Mode  Samples   Score  Score error  Units
c.b.MyBenchmark.compete      avgt       10  63.538        8.694   s/op

Process finished with exit code 0
```

Avg(*Collection.sort*)-Avg (this algo) (s)	**Total time(*Collection.sort*)-Total time(this algo) (s)**
-48.861 s	**10.06 minutes**

Merge sort

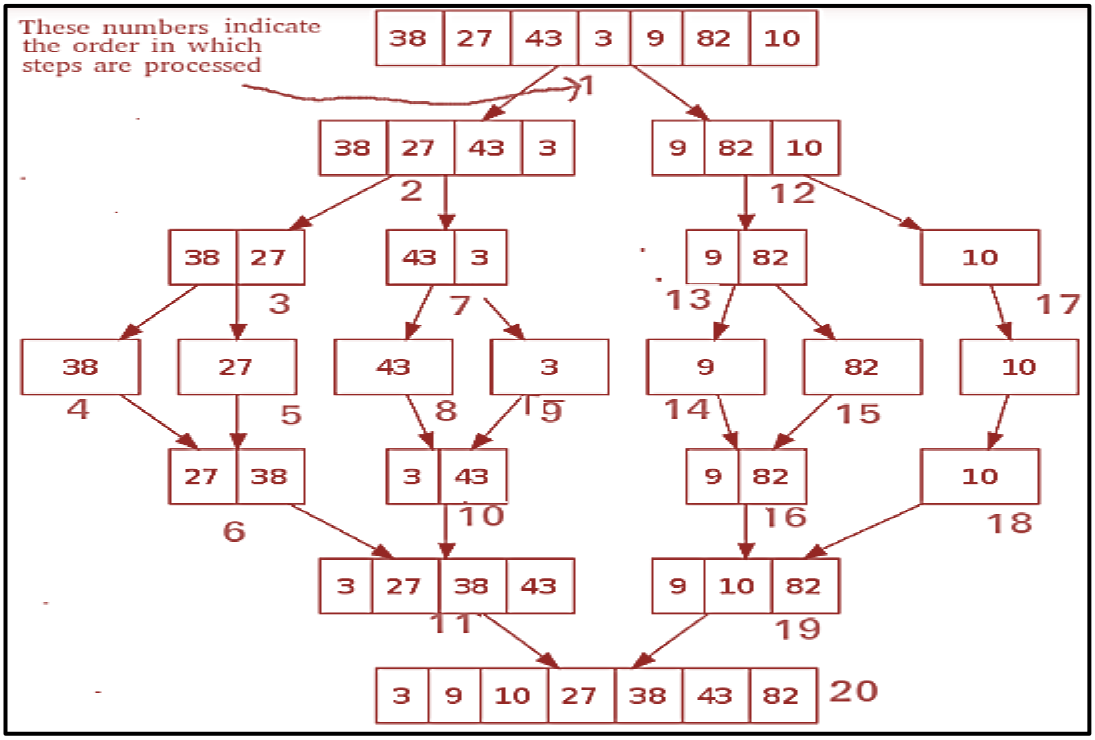
Merge Sort is a Divide and Conquer algorithm

Divide And Conquer

This technique can be divided into the following three parts:

1. **Divide:** This involves dividing the problem into some sub problem.
2. **Conquer:** Sub problem by calling recursively until sub problem solved.
3. **Combine:** The Sub problem Solved so that we will get find problem solution.

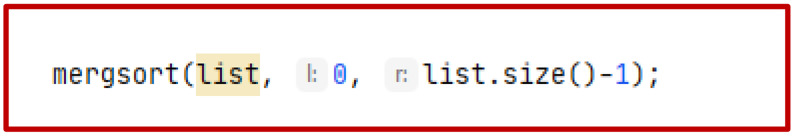
It divides the input array into two halves, calls itself for the two halves, and then merges the two sorted halves.



Worst and Average Case Time Complexity and best Case: Time complexity of Merge Sort is O(nLogn) in all 3 cases (worst, average and best) as merge sort always divides the array into two halves and takes linear time to merge two halves.

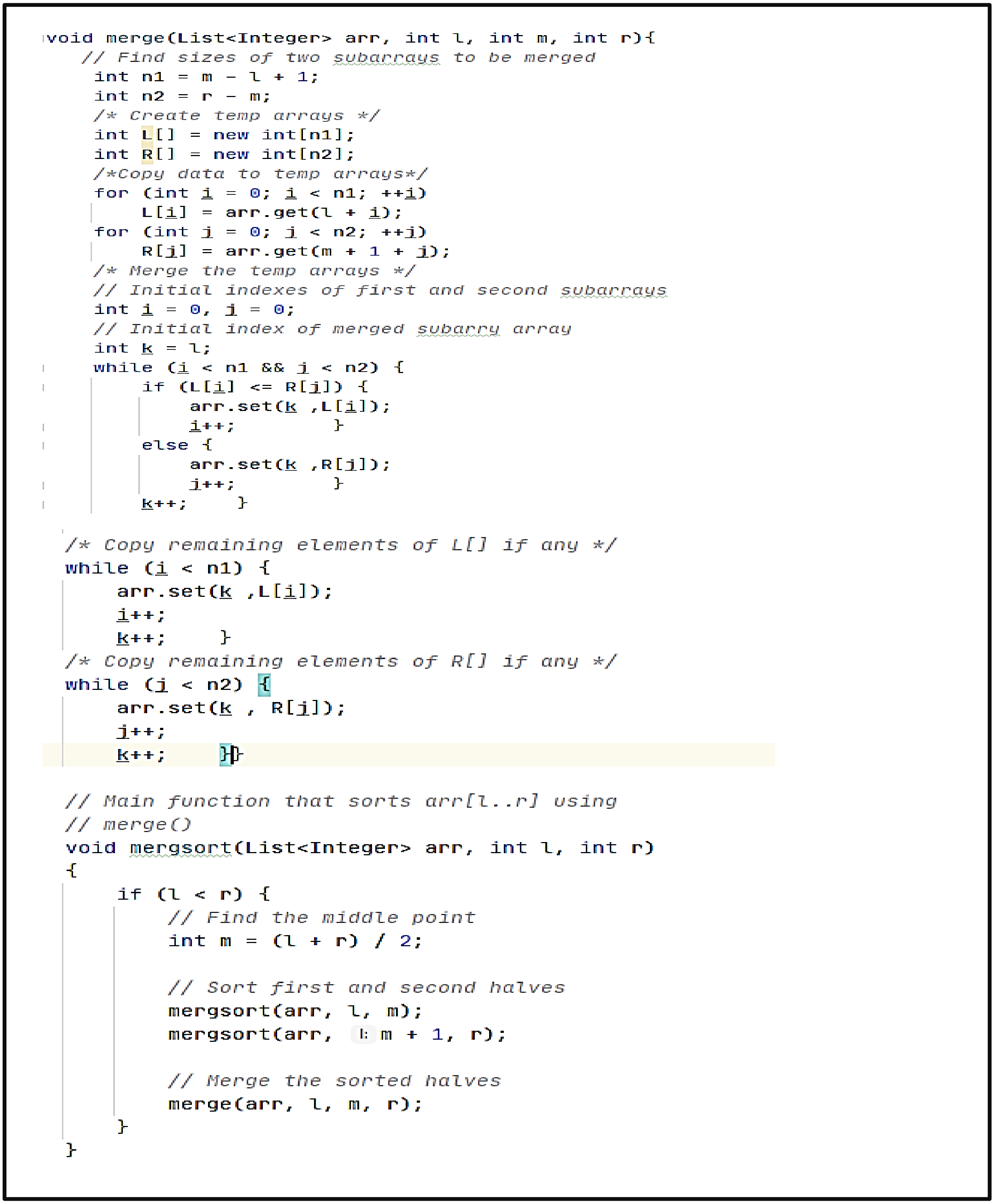
Implementation :

Inside sort(List list)method =>



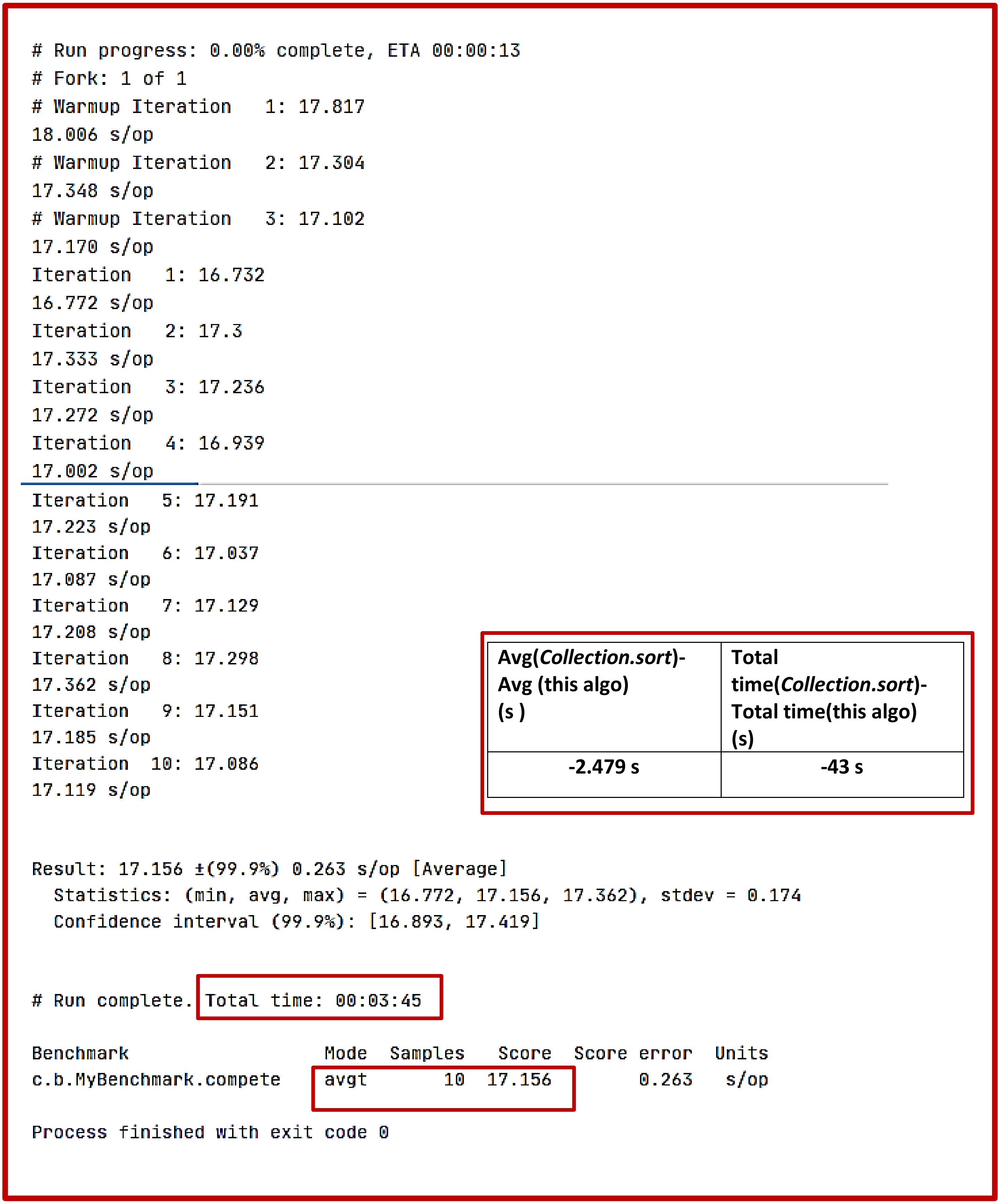
```
mergsort(list, l: 0, r: list.size()-1);
```

Functions :



```
void merge(List<Integer> arr, int l, int m, int r){
   // Find sizes of two subarrays to be merged
    int n1 = m - l + 1;
    int n2 = r - m;
    /* Create temp arrays */
    int L[] = new int[n1];
    int R[] = new int[n2];
    /*Copy data to temp arrays*/
    for (int i = 0; i < n1; ++i)
        L[i] = arr.get(l + i);
    for (int j = 0; j < n2; ++j)
        R[j] = arr.get(m + 1 + j);
    /* Merge the temp arrays */
    // Initial indexes of first and second subarrays
    int i = 0, j = 0;
    // Initial index of merged subarry array
    int k = l;
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            arr.set(k ,L[i]);
            i++;          }
        else {
            arr.set(k ,R[j]);
            j++;          }
        k++;     }

  /* Copy remaining elements of L[] if any */
  while (i < n1) {
      arr.set(k ,L[i]);
      i++;
      k++;     }
  /* Copy remaining elements of R[] if any */
  while (j < n2) {
      arr.set(k , R[j]);
      j++;
      k++;     }}

  // Main function that sorts arr[l..r] using
  // merge()
  void mergsort(List<Integer> arr, int l, int r)
  {
      if (l < r) {
          // Find the middle point
          int m = (l + r) / 2;

          // Sort first and second halves
          mergsort(arr, l, m);
          mergsort(arr, l: m + 1, r);

          // Merge the sorted halves
          merge(arr, l, m, r);
      }
  }
```

Results :



```
# Run progress: 0.00% complete, ETA 00:00:13
# Fork: 1 of 1
# Warmup Iteration   1: 17.817
18.006 s/op
# Warmup Iteration   2: 17.304
17.348 s/op
# Warmup Iteration   3: 17.102
17.170 s/op
Iteration   1: 16.732
16.772 s/op
Iteration   2: 17.3
17.333 s/op
Iteration   3: 17.236
17.272 s/op
Iteration   4: 16.939
17.002 s/op
Iteration   5: 17.191
17.223 s/op
Iteration   6: 17.037
17.087 s/op
Iteration   7: 17.129
17.208 s/op
Iteration   8: 17.298
17.362 s/op
Iteration   9: 17.151
17.185 s/op
Iteration  10: 17.086
17.119 s/op


Result: 17.156 ±(99.9%) 0.263 s/op [Average]
  Statistics: (min, avg, max) = (16.772, 17.156, 17.362), stdev = 0.174
  Confidence interval (99.9%): [16.893, 17.419]


# Run complete. Total time: 00:03:45

Benchmark                      Mode  Samples   Score  Score error  Units
c.b.MyBenchmark.compete        avgt       10  17.156        0.263   s/op

Process finished with exit code 0
```

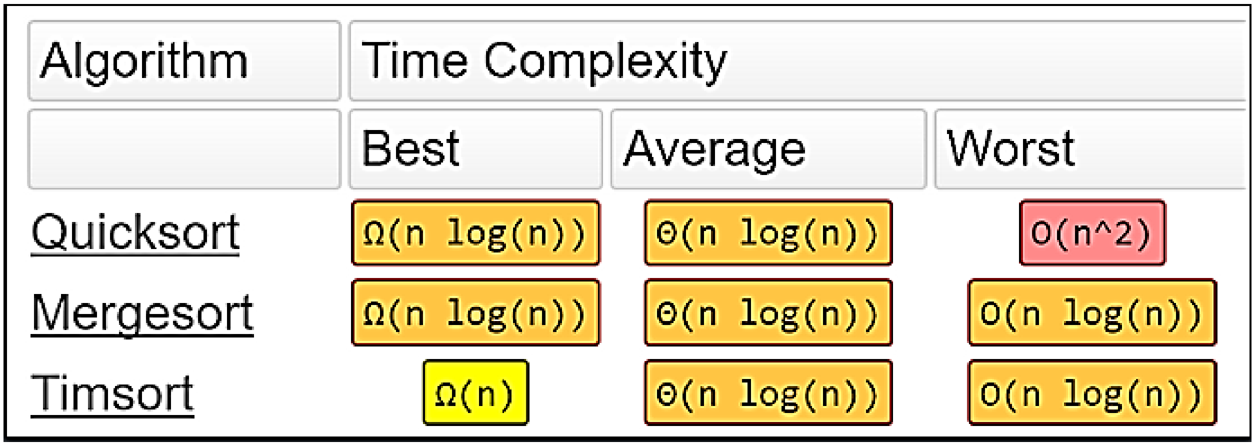
Avg(*Collection.sort*)-Avg (this algo) (s)	**Total time(*Collection.sort*)-Total time(this algo) (s)**
-2.479 s	**-43 s**

Collection.sort() :

the current Java versions use **TimSort** inside collection.sort().

TimSort is a sorting algorithm based **on Insertion Sort and Merge Sort**.

Timsort's sorting time is the same as Mergesort, which is faster than most of the other sorts you might know. Timsort actually makes use of Insertion sort and Mergesort.



Algorithm	Time Complexity		
	Best	Average	Worst
Quicksort	$\Omega(n \log(n))$	$\Theta(n \log(n))$	$O(n^2)$
Mergesort	$\Omega(n \log(n))$	$\Theta(n \log(n))$	$O(n \log(n))$
Timsort	$\Omega(n)$	$\Theta(n \log(n))$	$O(n \log(n))$

Used in Java's Arrays.sort() and Collections.sort(),as well as Python's sorted() and sort().

Timsort is a hybrid stable sorting algorithm ,Timsort was designed to take advantage of runs of consecutive ordered elements that already exist in most real-world data, natural runs.

A **"Hybrid algorithm"** is an algorithm that combines two or more other algorithms that solve the same problem , so TimSort is hybrid algorithm because it is based on Insertion Sort and Merge Sort.

A **"stable sorting algorithm "** is an algorithm that maintain the relative order of records with equal keys (i.e. order of elements with same key is kept) so to maintain stability we should not exchange 2 numbers of equal value. This not only keeps their original positions in the list but enables the algorithm to be faster.

Steps in TimSort :

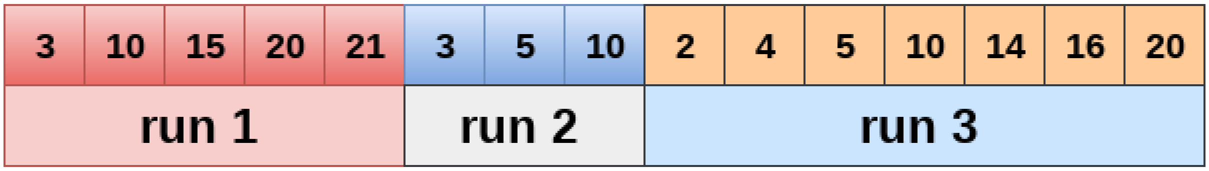
1-Divide the array into the number of blocks known as run.

2-Consider size of run may vary from 32 to 64 depending upon the size of the array

3-sort those runs using insertion sort one by one (If the size of Array is less than run, then Array get sorted just by using Insertion Sort)

4-merge those runs using combine function used in merge sort.(Note that merge function performs well when sizes sub-arrays are powers of 2)

5-Double the size of merged sub-arrays after every iteration.



3	10	15	20	21	3	5	10	2	4	5	10	14	16	20
run 1					run 2			run 3						

Why insertion sort not other sorting algorithm ?

An insertion sort is a simple sort which is most effective on small list. It is quite slow at larger lists, but very fast with small lists

1-Look at elements one by one.

2-Build up sorted list by inserting the element at the correct location.

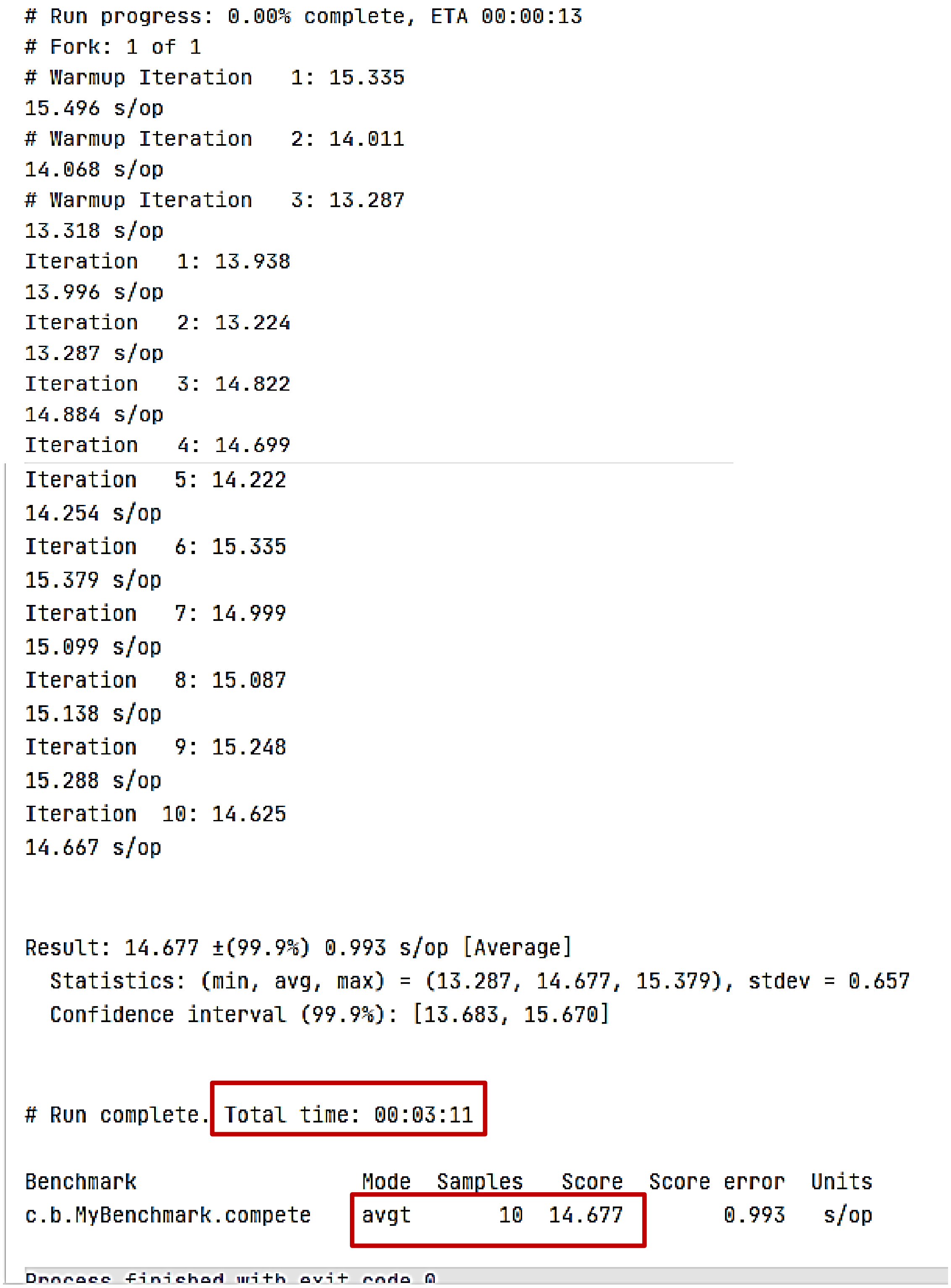
Why Timsort (collection.sort)based on merge sort not Quick sort sice Quicksort is typically over twice as fast as merge sort.?

The API guarantees a stable sorting which Quicksort doesn’t offer , but Quicksort can used for primitive arrays , The efficiency advantage of Quicksort is needing less memory when done in-place and behaves well even withcaching and virtual memory But it has a worst case performance when the input is already sorted , which it is best case in TimSort .

In the worst case, Timsort takes O (n log n) comparisons to sort an array of n elements. **In the best case**, which occurs when the input is already sorted, it runs in linear time, meaning that it is an adaptive sorting algorithm.

It is advantageous over Quicksort for sorting object references (objects with different identity)because these require expensive memory indirection to access data and perform comparisons to change their order, Therefore, Quicksort is not an option. and Quicksort's cache coherence benefits are greatly reduced.

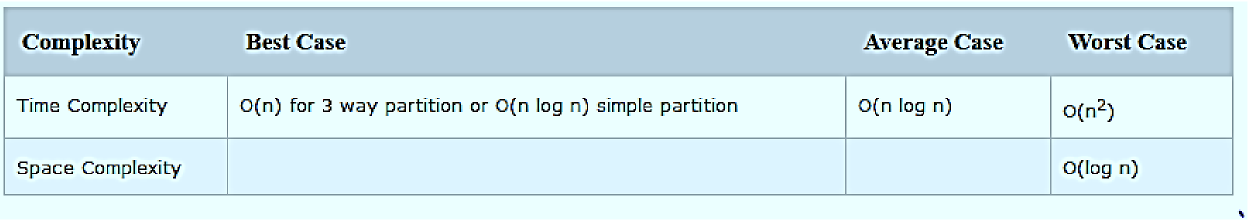
Results :



```
# Run progress: 0.00% complete, ETA 00:00:13
# Fork: 1 of 1
# Warmup Iteration   1: 15.335
15.496 s/op
# Warmup Iteration   2: 14.011
14.068 s/op
# Warmup Iteration   3: 13.287
13.318 s/op
Iteration   1: 13.938
13.996 s/op
Iteration   2: 13.224
13.287 s/op
Iteration   3: 14.822
14.884 s/op
Iteration   4: 14.699
Iteration   5: 14.222
14.254 s/op
Iteration   6: 15.335
15.379 s/op
Iteration   7: 14.999
15.099 s/op
Iteration   8: 15.087
15.138 s/op
Iteration   9: 15.248
15.288 s/op
Iteration  10: 14.625
14.667 s/op


Result: 14.677 ±(99.9%) 0.993 s/op [Average]
  Statistics: (min, avg, max) = (13.287, 14.677, 15.379), stdev = 0.657
  Confidence interval (99.9%): [13.683, 15.670]


# Run complete. Total time: 00:03:11

Benchmark                    Mode  Samples   Score  Score error  Units
c.b.MyBenchmark.compete      avgt       10  14.677        0.993   s/op

Process finished with exit code 0
```

QuickSort (partition-exchange sort)

Like Merge Sort, QuickSort is a Divide and Conquer algorithm , It picks an element as pivot and partitions the given array around the picked pivot, in efficient implementation it is usually not stable sort.



Complexity	Best Case	Average Case	Worst Case
Time Complexity	O(n) for 3 way partition or O(n log n) simple partition	O(n log n)	$O(n^2)$
Space Complexity			O(log n)

The key process in quickSort is partition(). Target of partitions is, given an array and an element x of array as pivot, put this pivot at its correct position in sorted array and put all smaller elements (smaller than pivot) before pivot, and put all greater elements (greater than pivot) after pivotx. All this should be done in linear time. So It first divides the input array into two smaller sub-arrays, the low elements and the high elements. It then recursively sorts the sub-arrays

Quick sort on average takes O(nlogn) comparisons to sort n items . in worst case it makes O(n2)comparisons though this behavior is rare .
If last element or first element is always picked as pivot, the worst case would occur when the array is already sorted in increasing or decreasing order , or when all elements of array are equal . This will makes unbalanced partition as the pivot divides the array into two sub_array of sizes 0 and n-1 . if this happens in ever partition then each recursive call a list of size one less than the previous list and this will result in O(n2)time [T(n)=T(n-1)+cn = O(n2)].

If the pivot divides the array into two nearly equal pieces so each recursive call processes alist of half the size, the best case would occur and this will result in O(nlogn) time [T(n)= 2 T(n/2) + cn = O(n log(n))].

Three steps to involve all processes for Quicksort are:

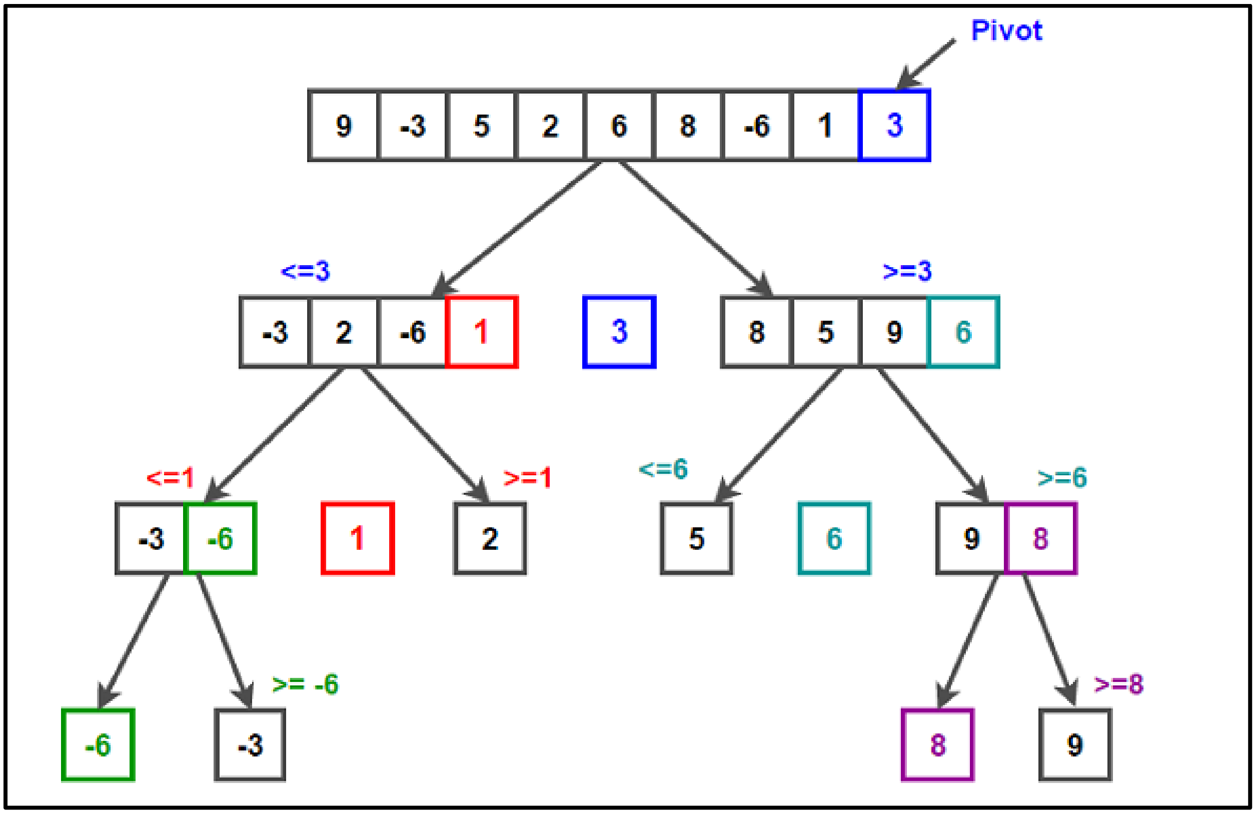
1. Pick a *pivot*, from the array(first element or last element or random index).
2. *Partitioning*: reorder the array so that all elements with values less than the pivot come before the pivot, while all elements with values greater than the pivot come after it (equal values can go either way). After this partitioning, the pivot is in its final position. This is called the *partition* operation.
3. Recursively apply the above steps to the sub-array of elements with smaller values and separately to the sub-array of elements with greater values.(The base case of the recursion is arrays of size 1, which are in order by definition, so they never need to be sorted.)

Lomuto partition scheme quick sort

standard quick sort

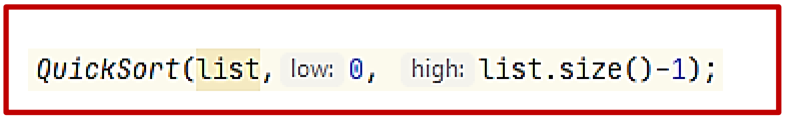
last element in the array as pivot . (or first element)

This scheme degrades to $O(n^2)$ when the array is already in order.(worst case)



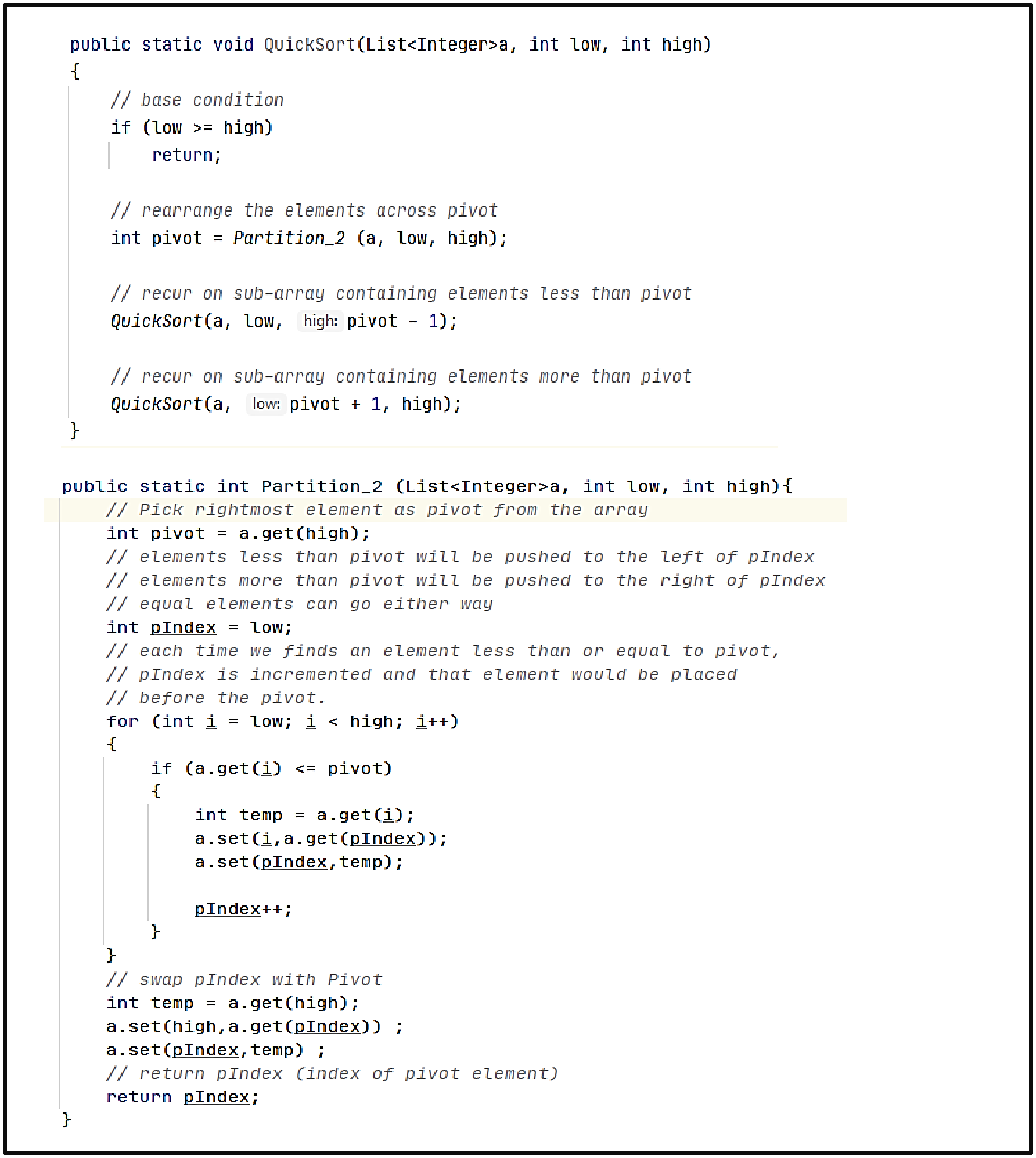
Implementation :

Inside sort(List list)method =>



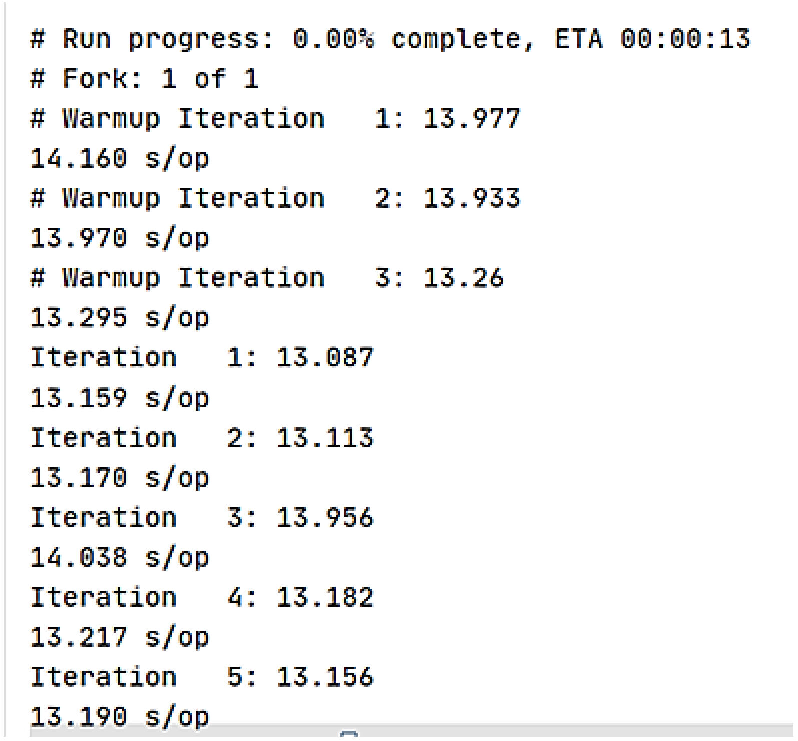
```
QuickSort(list, low: 0, high: list.size()-1);
```

Functions :

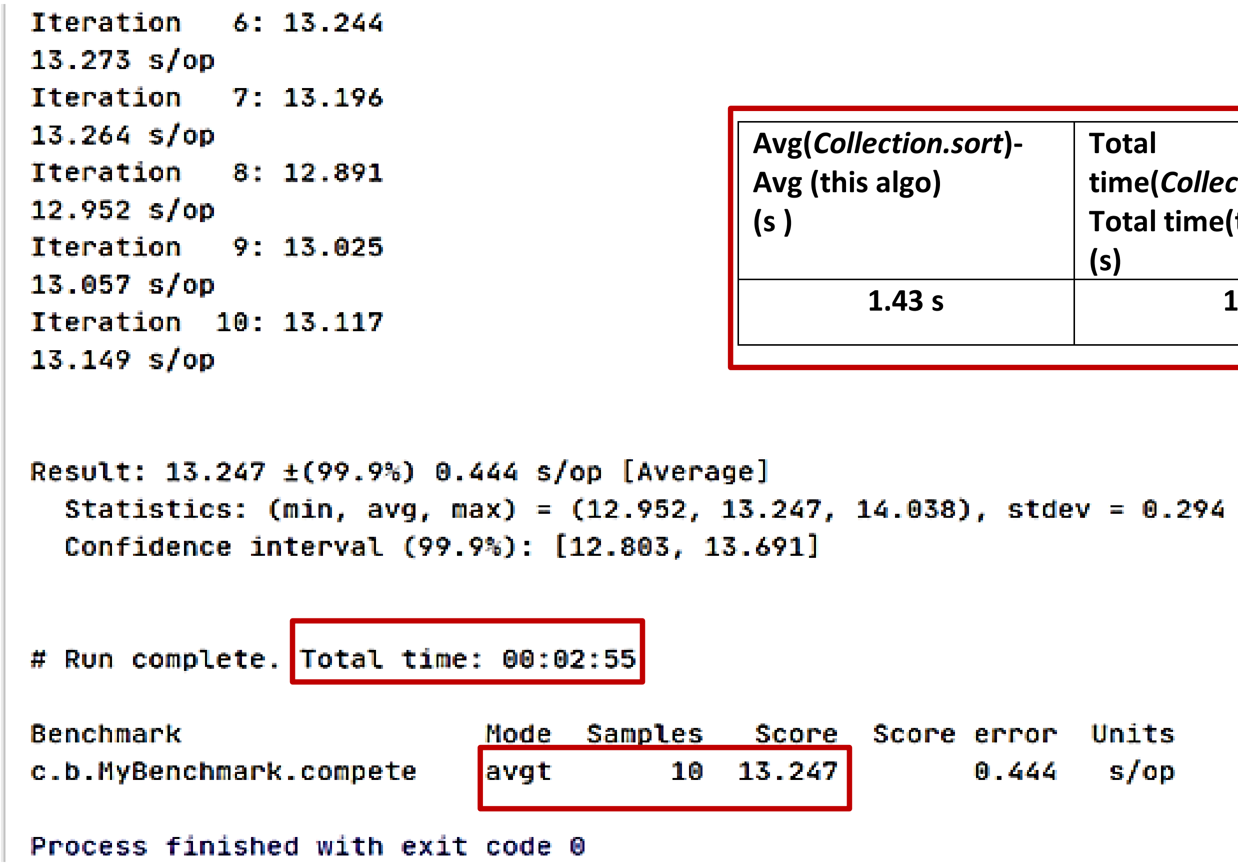


```
public static void QuickSort(List<Integer>a, int low, int high)
{
    // base condition
    if (low >= high)
        return;

    // rearrange the elements across pivot
    int pivot = Partition_2 (a, low, high);

    // recur on sub-array containing elements less than pivot
    QuickSort(a, low, high: pivot - 1);

    // recur on sub-array containing elements more than pivot
    QuickSort(a, low: pivot + 1, high);
}

public static int Partition_2 (List<Integer>a, int low, int high){
    // Pick rightmost element as pivot from the array
    int pivot = a.get(high);
    // elements less than pivot will be pushed to the left of pIndex
    // elements more than pivot will be pushed to the right of pIndex
    // equal elements can go either way
    int pIndex = low;
    // each time we finds an element less than or equal to pivot,
    // pIndex is incremented and that element would be placed
    // before the pivot.
    for (int i = low; i < high; i++)
    {
        if (a.get(i) <= pivot)
        {
            int temp = a.get(i);
            a.set(i,a.get(pIndex));
            a.set(pIndex,temp);

            pIndex++;
        }
    }
    // swap pIndex with Pivot
    int temp = a.get(high);
    a.set(high,a.get(pIndex)) ;
    a.set(pIndex,temp) ;
    // return pIndex (index of pivot element)
    return pIndex;
}
```

Results:



```
# Run progress: 0.00% complete, ETA 00:00:13
# Fork: 1 of 1
# Warmup Iteration   1: 13.977
14.160 s/op
# Warmup Iteration   2: 13.933
13.970 s/op
# Warmup Iteration   3: 13.26
13.295 s/op
Iteration   1: 13.087
13.159 s/op
Iteration   2: 13.113
13.170 s/op
Iteration   3: 13.956
14.038 s/op
Iteration   4: 13.182
13.217 s/op
Iteration   5: 13.156
13.190 s/op
```



```
Iteration   6: 13.244
13.273 s/op
Iteration   7: 13.196
13.264 s/op
Iteration   8: 12.891
12.952 s/op
Iteration   9: 13.025
13.057 s/op
Iteration  10: 13.117
13.149 s/op


Result: 13.247 ±(99.9%) 0.444 s/op [Average]
  Statistics: (min, avg, max) = (12.952, 13.247, 14.038), stdev = 0.294
  Confidence interval (99.9%): [12.803, 13.691]


# Run complete. Total time: 00:02:55

Benchmark                          Mode   Samples    Score  Score error   Units
c.b.MyBenchmark.compete            avgt        10   13.247        0.444    s/op

Process finished with exit code 0
```

Avg(*Collection.sort*)-Avg (this algo) (s)	**Total time(*Collection.sort*)-Total time(this algo) (s)**
1.43 s	**16 s**

Randomized Quick Sort scheme

Since quick sort normally choose the last element or first element as pivot, the worst case would occur on sorted or nearly sorted array , the problem can be solved by choosing a random index of a pivot or choose median of first , middle ,last element for the pivot . so I tried using a random index of a pivot(**Randomized Quick Sort**)

Random element in the array as pivot (Partition around a random element) ,

No specific input elicits the worst-casebehavior , The worst case is determined only by the output of a random-number generator . that this will happen and trigger the worst-case behavior.

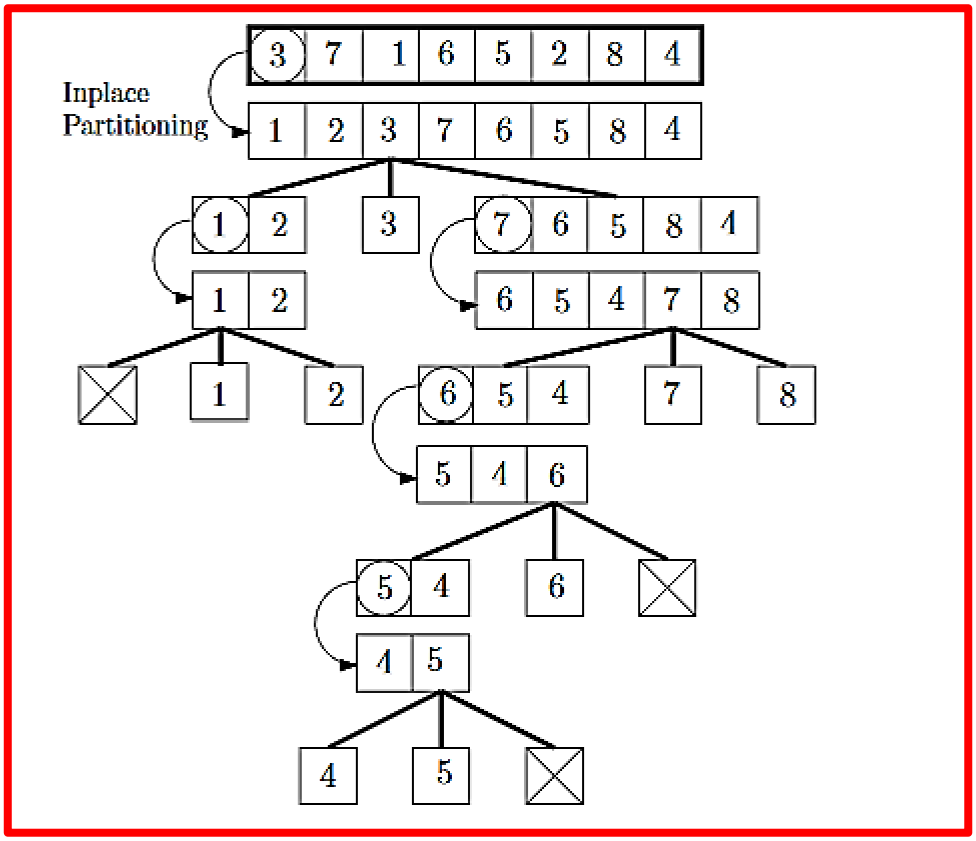
The advantage of randomized quicksort is that there's no one input that will always cause it to run in time O(n log n) and the runtime is expected to be O(n log n). Deterministic quicksort algorithms usually have the drawback that either (1) they run in worst-case time O(n log n), but with a high constant factor, or (2) they run in worst-case time $O(n^2)$ and the sort of input that triggers this case is deterministic.

In summary, randomized quicksort is cool because:

-quicksort is often fastest in practice

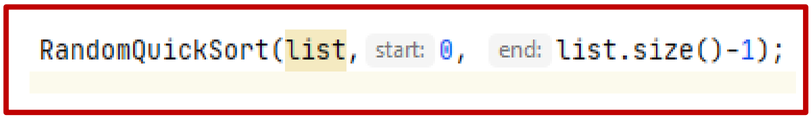
-it has *O*(*n*log*n*) expected run time, regardless of the input distribution

-the same step of standard quick sort above but with random pivot .



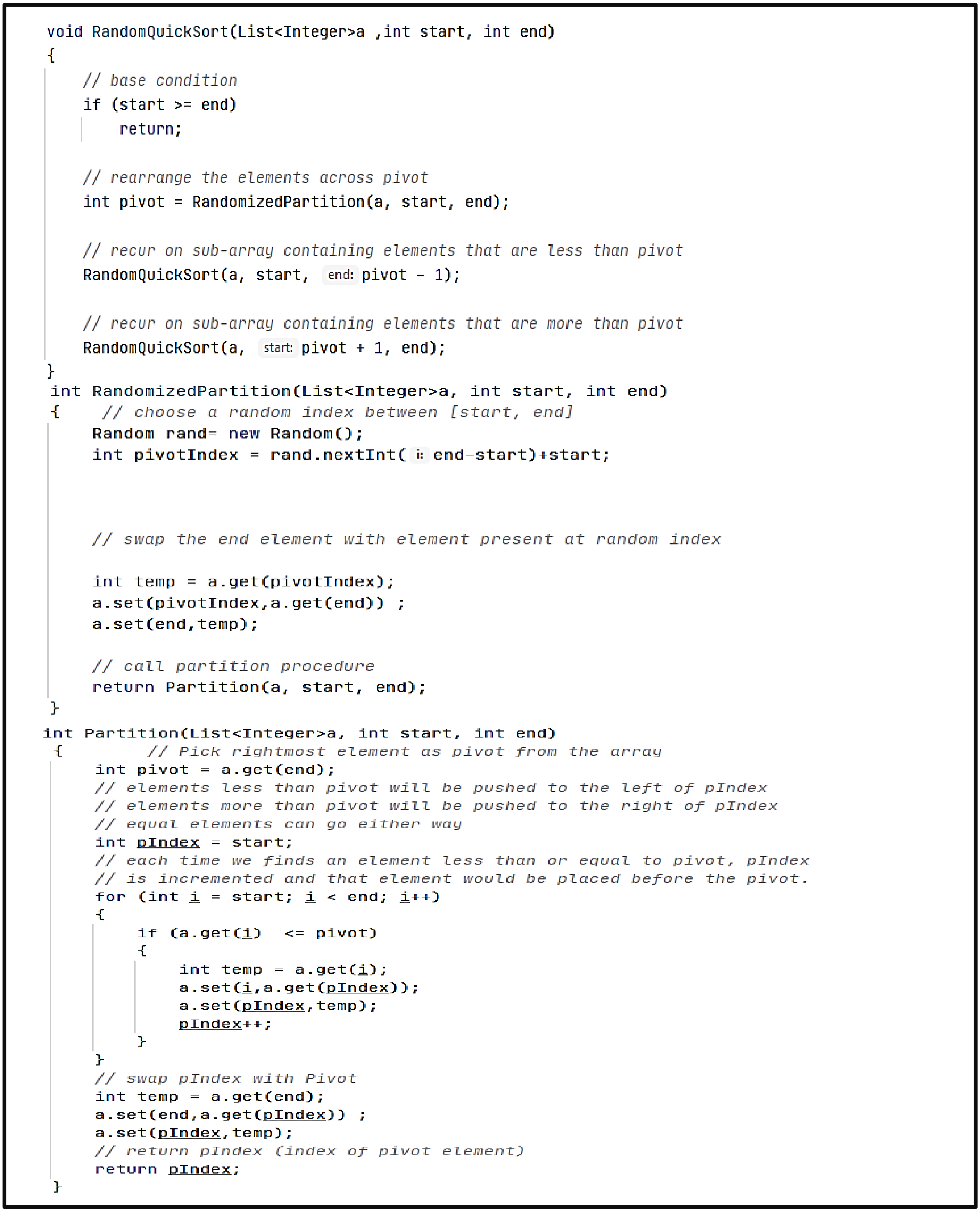
Implementation :

Inside sort(List list)method =>



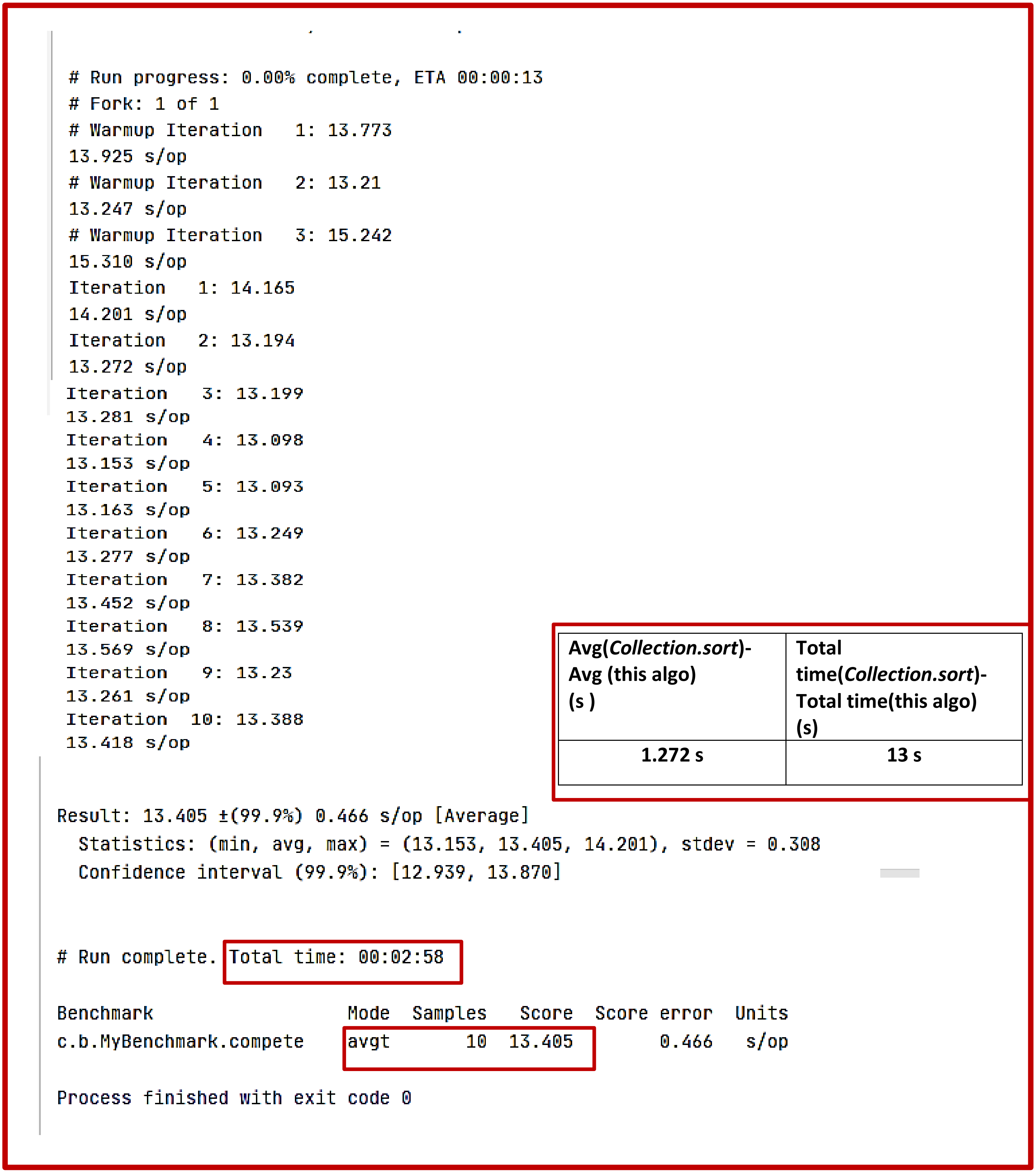
```
RandomQuickSort(list, start: 0, end: list.size()-1);
```

Functions :



```
void RandomQuickSort(List<Integer>a ,int start, int end)
{
    // base condition
    if (start >= end)
        return;

    // rearrange the elements across pivot
    int pivot = RandomizedPartition(a, start, end);

    // recur on sub-array containing elements that are less than pivot
    RandomQuickSort(a, start, end: pivot - 1);

    // recur on sub-array containing elements that are more than pivot
    RandomQuickSort(a, start: pivot + 1, end);
}
int RandomizedPartition(List<Integer>a, int start, int end)
{    // choose a random index between [start, end]
    Random rand= new Random();
    int pivotIndex = rand.nextInt( i: end-start)+start;



    // swap the end element with element present at random index

    int temp = a.get(pivotIndex);
    a.set(pivotIndex,a.get(end)) ;
    a.set(end,temp);

    // call partition procedure
    return Partition(a, start, end);
}
int Partition(List<Integer>a, int start, int end)
{        // Pick rightmost element as pivot from the array
    int pivot = a.get(end);
    // elements less than pivot will be pushed to the left of pIndex
    // elements more than pivot will be pushed to the right of pIndex
    // equal elements can go either way
    int pIndex = start;
    // each time we finds an element less than or equal to pivot, pIndex
    // is incremented and that element would be placed before the pivot.
    for (int i = start; i < end; i++)
    {
        if (a.get(i)  <= pivot)
        {
            int temp = a.get(i);
            a.set(i,a.get(pIndex));
            a.set(pIndex,temp);
            pIndex++;
        }
    }
    // swap pIndex with Pivot
    int temp = a.get(end);
    a.set(end,a.get(pIndex)) ;
    a.set(pIndex,temp);
    // return pIndex (index of pivot element)
    return pIndex;
}
```

Results :



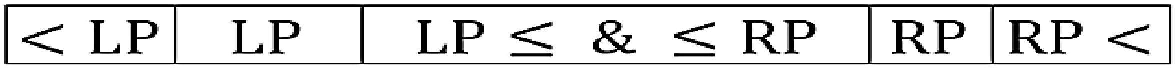
```
# Run progress: 0.00% complete, ETA 00:00:13
# Fork: 1 of 1
# Warmup Iteration   1: 13.773
13.925 s/op
# Warmup Iteration   2: 13.21
13.247 s/op
# Warmup Iteration   3: 15.242
15.310 s/op
Iteration   1: 14.165
14.201 s/op
Iteration   2: 13.194
13.272 s/op
Iteration   3: 13.199
13.281 s/op
Iteration   4: 13.098
13.153 s/op
Iteration   5: 13.093
13.163 s/op
Iteration   6: 13.249
13.277 s/op
Iteration   7: 13.382
13.452 s/op
Iteration   8: 13.539
13.569 s/op
Iteration   9: 13.23
13.261 s/op
Iteration  10: 13.388
13.418 s/op


Result: 13.405 ±(99.9%) 0.466 s/op [Average]
  Statistics: (min, avg, max) = (13.153, 13.405, 14.201), stdev = 0.308
  Confidence interval (99.9%): [12.939, 13.870]


# Run complete. Total time: 00:02:58

Benchmark                    Mode  Samples   Score  Score error  Units
c.b.MyBenchmark.compete      avgt       10  13.405        0.466   s/op

Process finished with exit code 0
```

Avg(*Collection.sort*)-Avg (this algo) (s)	**Total time(*Collection.sort*)-Total time(this algo) (s)**
1.272 s	**13 s**

Dual pivot Quicksort

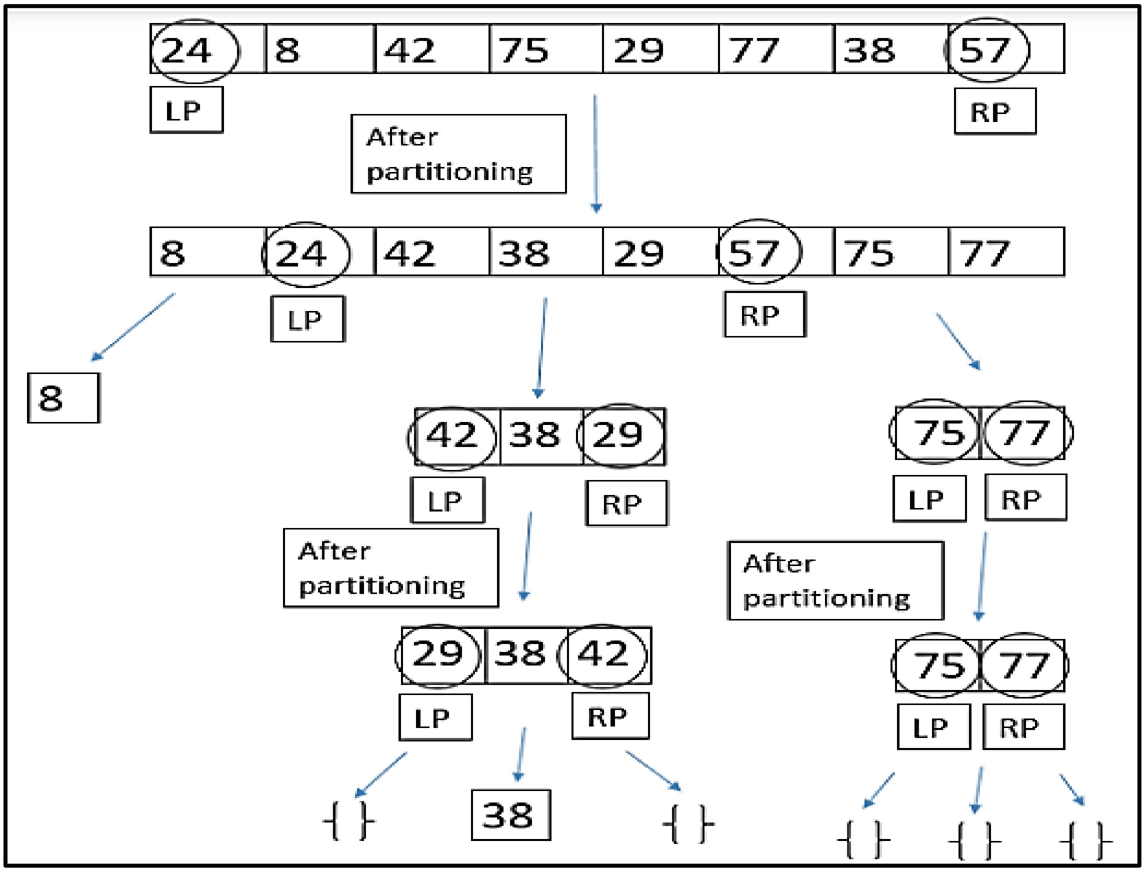
The idea of dual pivot quick sort is to take two pivots, one in the left end of the array and the second, in the right end of the array. The left pivot must be less than or equal to the right pivot, so we swap them if necessary.

Then, we begin partitioning the array into three parts: in the first part, all elements will be less than the left pivot, in the second part all elements will be greater or equal to the left pivot and also will be less than or equal to the right pivot, and in the third part all elements will be greater than the right pivot. Then, we shift the two pivots to their appropriate positions as we see in the below bar, and after that we begin quicksorting these three parts recursively, using this method.



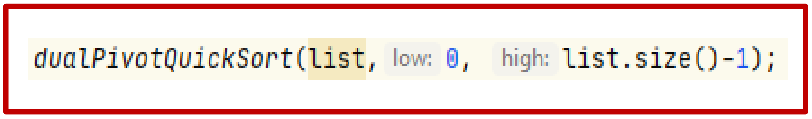
$< LP$	LP	$LP \leq \ \& \ \leq RP$	RP	$RP <$

But still, the worst case will remain $O(n^2)$ when the array is already sorted in an increasing or decreasing order.



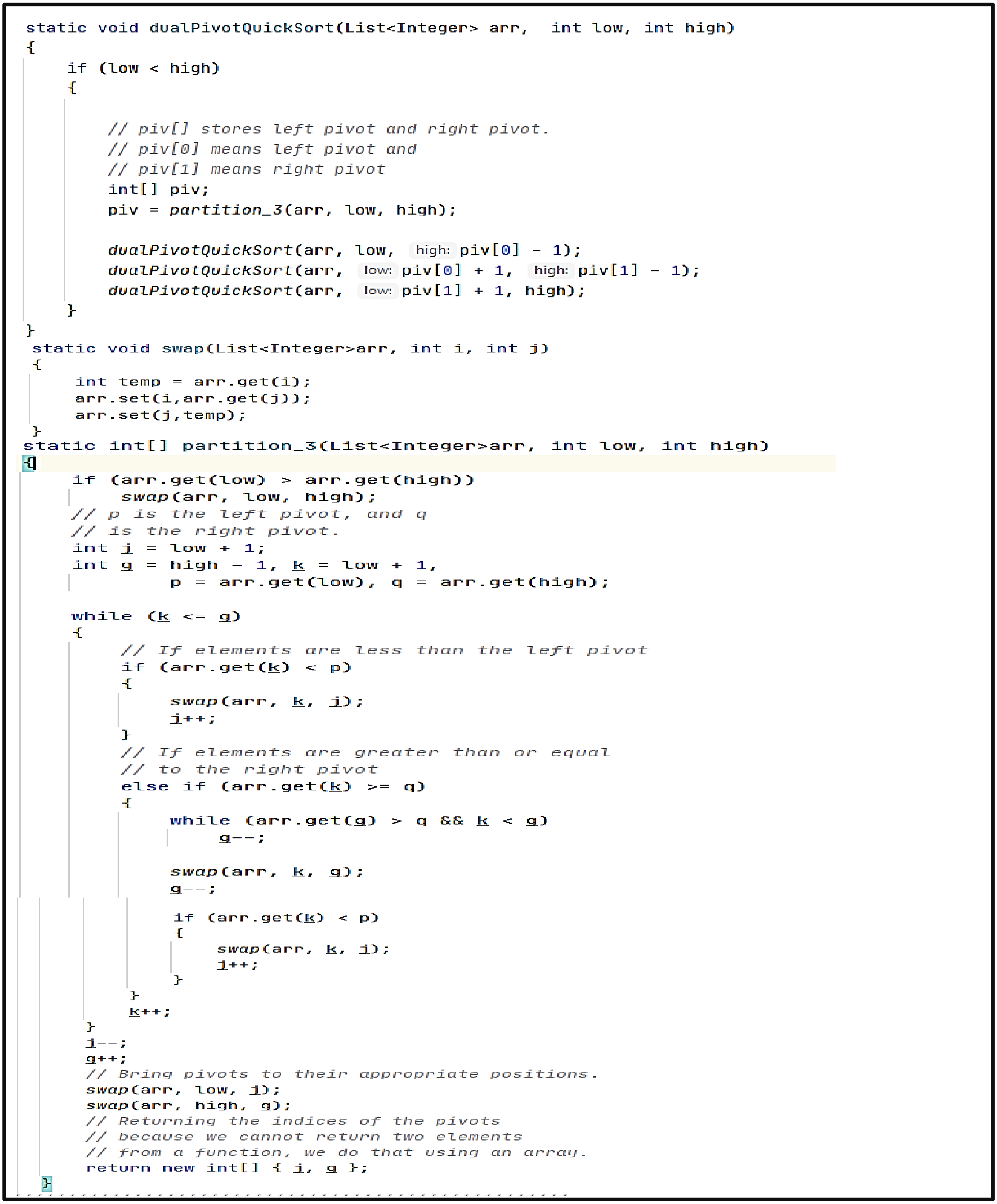
Implementation :

Inside sort(List list)method =>



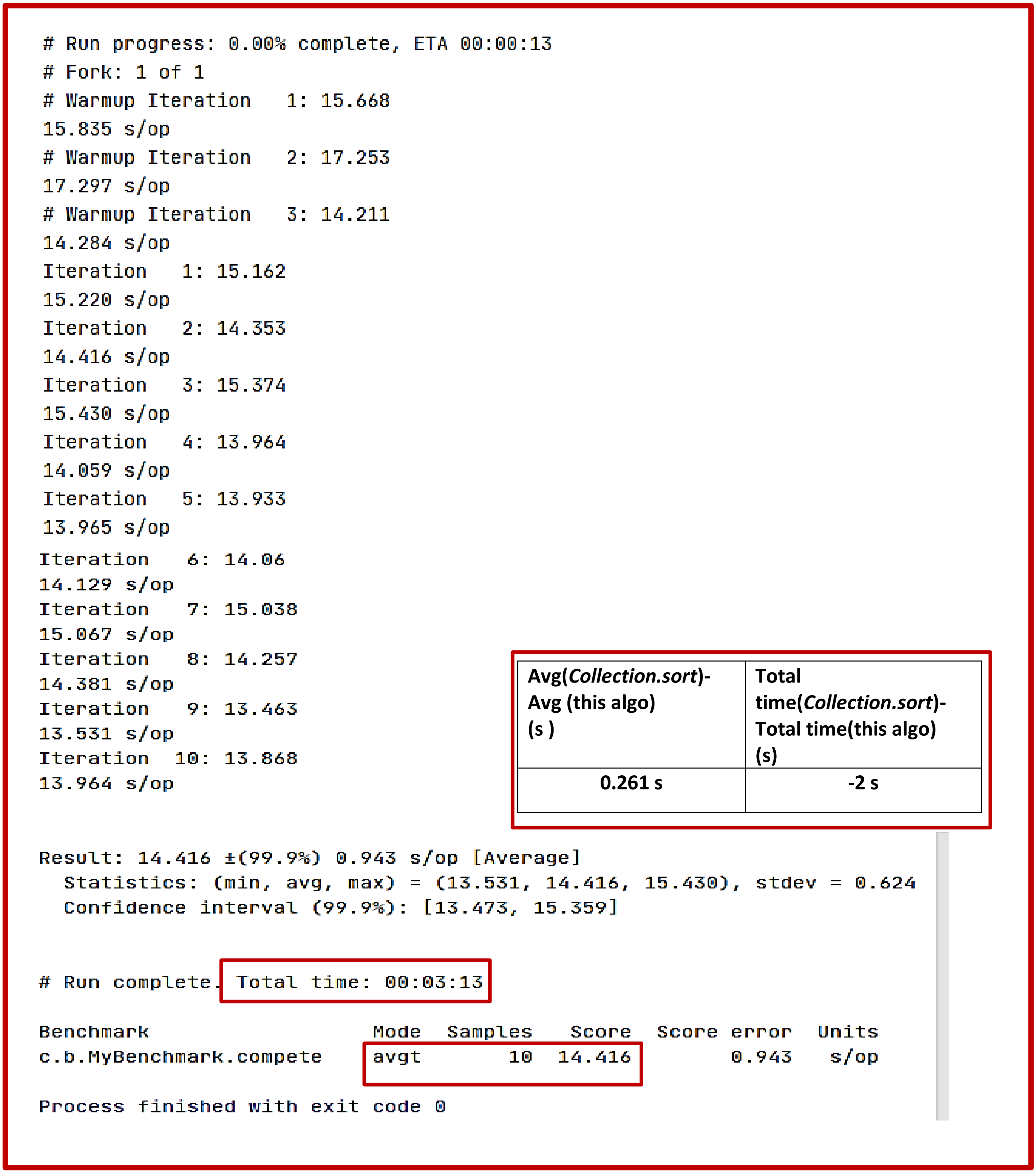
```
dualPivotQuickSort(list, low: 0, high: list.size()-1);
```

Functions :



```
static void dualPivotQuickSort(List<Integer> arr,  int low, int high)
{
    if (low < high)
    {

        // piv[] stores left pivot and right pivot.
        // piv[0] means left pivot and
        // piv[1] means right pivot
        int[] piv;
        piv = partition_3(arr, low, high);

        dualPivotQuickSort(arr, low, high: piv[0] - 1);
        dualPivotQuickSort(arr, low: piv[0] + 1, high: piv[1] - 1);
        dualPivotQuickSort(arr, low: piv[1] + 1, high);
    }
}
 static void swap(List<Integer>arr, int i, int j)
 {
     int temp = arr.get(i);
     arr.set(i,arr.get(j));
     arr.set(j,temp);
 }
static int[] partition_3(List<Integer>arr, int low, int high)
{
     if (arr.get(low) > arr.get(high))
         swap(arr, low, high);
     // p is the left pivot, and q
     // is the right pivot.
     int j = low + 1;
     int g = high - 1, k = low + 1,
             p = arr.get(low), q = arr.get(high);

     while (k <= g)
     {
         // If elements are less than the left pivot
         if (arr.get(k) < p)
         {
             swap(arr, k, j);
             j++;
         }
         // If elements are greater than or equal
         // to the right pivot
         else if (arr.get(k) >= q)
         {
             while (arr.get(g) > q && k < g)
                 g--;

             swap(arr, k, g);
             g--;

             if (arr.get(k) < p)
             {
                 swap(arr, k, j);
                 j++;
             }
         }
         k++;
     }
     j--;
     g++;
     // Bring pivots to their appropriate positions.
     swap(arr, low, j);
     swap(arr, high, g);
     // Returning the indices of the pivots
     // because we cannot return two elements
     // from a function, we do that using an array.
     return new int[] { j, g };
}
```

Results :



```
# Run progress: 0.00% complete, ETA 00:00:13
# Fork: 1 of 1
# Warmup Iteration   1: 15.668
15.835 s/op
# Warmup Iteration   2: 17.253
17.297 s/op
# Warmup Iteration   3: 14.211
14.284 s/op
Iteration   1: 15.162
15.220 s/op
Iteration   2: 14.353
14.416 s/op
Iteration   3: 15.374
15.430 s/op
Iteration   4: 13.964
14.059 s/op
Iteration   5: 13.933
13.965 s/op
Iteration   6: 14.06
14.129 s/op
Iteration   7: 15.038
15.067 s/op
Iteration   8: 14.257
14.381 s/op
Iteration   9: 13.463
13.531 s/op
Iteration  10: 13.868
13.964 s/op


Result: 14.416 ±(99.9%) 0.943 s/op [Average]
  Statistics: (min, avg, max) = (13.531, 14.416, 15.430), stdev = 0.624
  Confidence interval (99.9%): [13.473, 15.359]


# Run complete. Total time: 00:03:13

Benchmark                       Mode  Samples   Score  Score error  Units
c.b.MyBenchmark.compete         avgt       10  14.416        0.943   s/op

Process finished with exit code 0
```

Avg(*Collection.sort*)-Avg (this algo) (s)	Total time(*Collection.sort*)-Total time(this algo) (s)
0.261 s	**-2 s**

Tail Recursive Quick Sort

In QuickSort, partition function is in-place, but we need extra space (stack) for recursive function calls. in worst case requires O(n) space on function call stack.To make sure at most *O*(log *n*) space is used(Reducing worst case space to Log n) .

This idea is based on tail call elimination .(A recursive function is tail recursive when recursive call is the last thing executed by the function) . Recursion uses stack to keep track of function calls. With every function call, a new frame is pushed onto the stack which contains local variables and data of that call) so ,

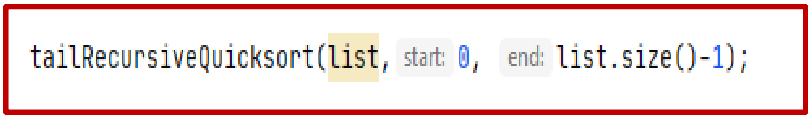
Tail call elimination reduces space complexity of recursion from O(N) to O(1). As function call is eliminated, no new stack frames are created and the function is executed in constant memory space.

In QuickSort Tail Call Optimization (Reducing worst case space from O(n) to O(Log n).by recur first into the smaller side of the partition, then use a tail call to recur into the other(recursive call only for the smaller part after partition.) , if left part becomes smaller, then we make recursive call for left part. Else for the right part.

By using a while loop and have reduced the number of recursive calls(minimize the recursive depth).

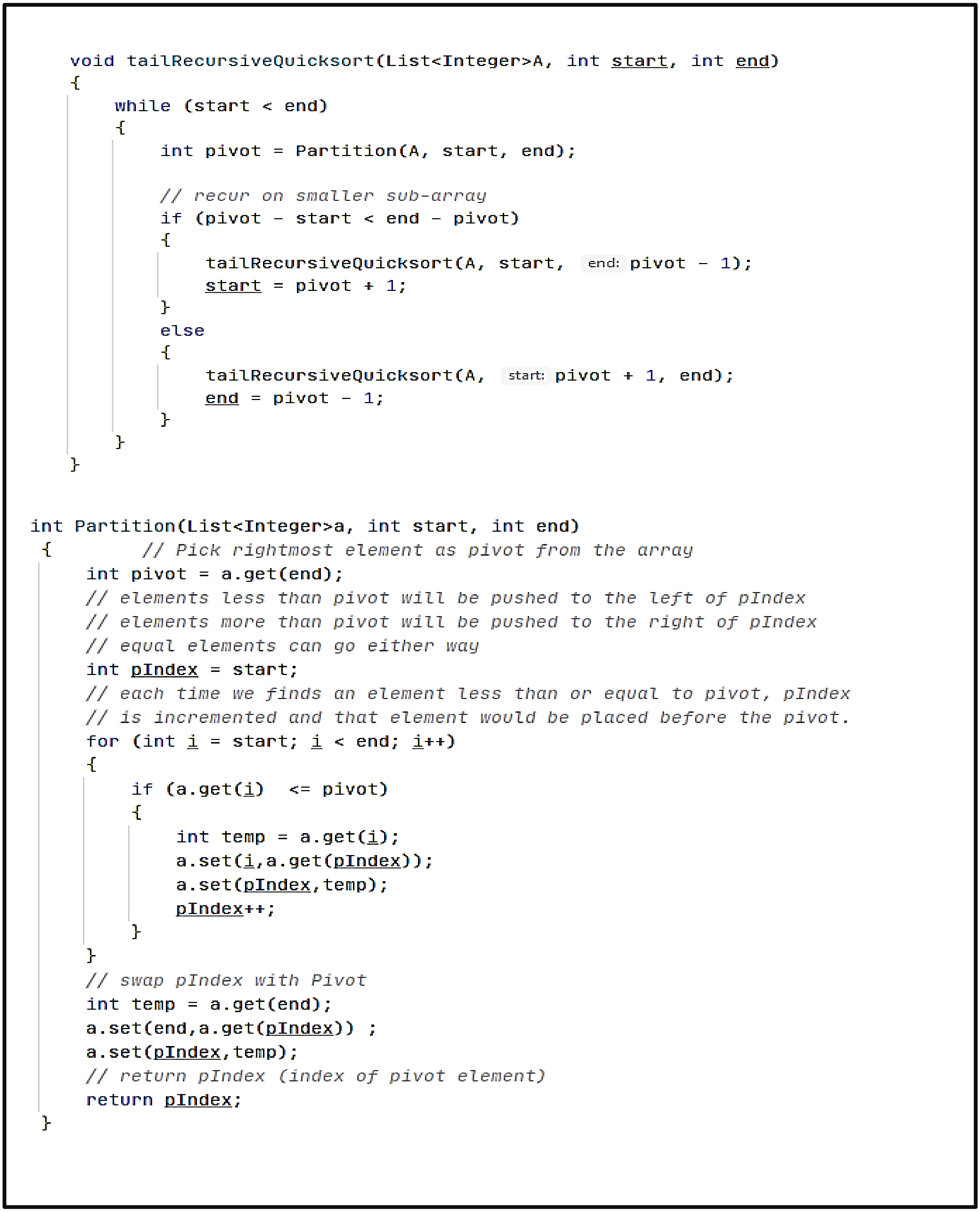
Implementation :

Inside sort(List list)method =>



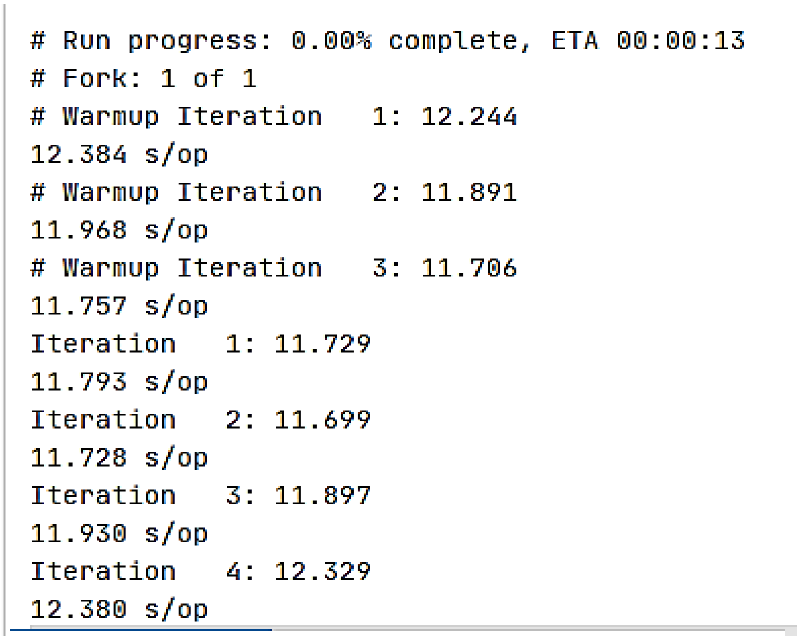
```
tailRecursiveQuicksort(list, start: 0, end: list.size()-1);
```

Functions :

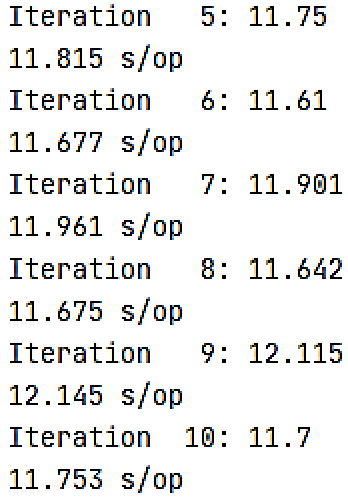


```
    void tailRecursiveQuicksort(List<Integer>A, int start, int end)
    {
        while (start < end)
        {
            int pivot = Partition(A, start, end);

            // recur on smaller sub-array
            if (pivot - start < end - pivot)
            {
                tailRecursiveQuicksort(A, start, end: pivot - 1);
                start = pivot + 1;
            }
            else
            {
                tailRecursiveQuicksort(A, start: pivot + 1, end);
                end = pivot - 1;
            }
        }
    }


int Partition(List<Integer>a, int start, int end)
 {        // Pick rightmost element as pivot from the array
     int pivot = a.get(end);
     // elements less than pivot will be pushed to the left of pIndex
     // elements more than pivot will be pushed to the right of pIndex
     // equal elements can go either way
     int pIndex = start;
     // each time we finds an element less than or equal to pivot, pIndex
     // is incremented and that element would be placed before the pivot.
     for (int i = start; i < end; i++)
     {
         if (a.get(i)  <= pivot)
         {
             int temp = a.get(i);
             a.set(i,a.get(pIndex));
             a.set(pIndex,temp);
             pIndex++;
         }
     }
     // swap pIndex with Pivot
     int temp = a.get(end);
     a.set(end,a.get(pIndex)) ;
     a.set(pIndex,temp);
     // return pIndex (index of pivot element)
     return pIndex;
 }
```

Results :

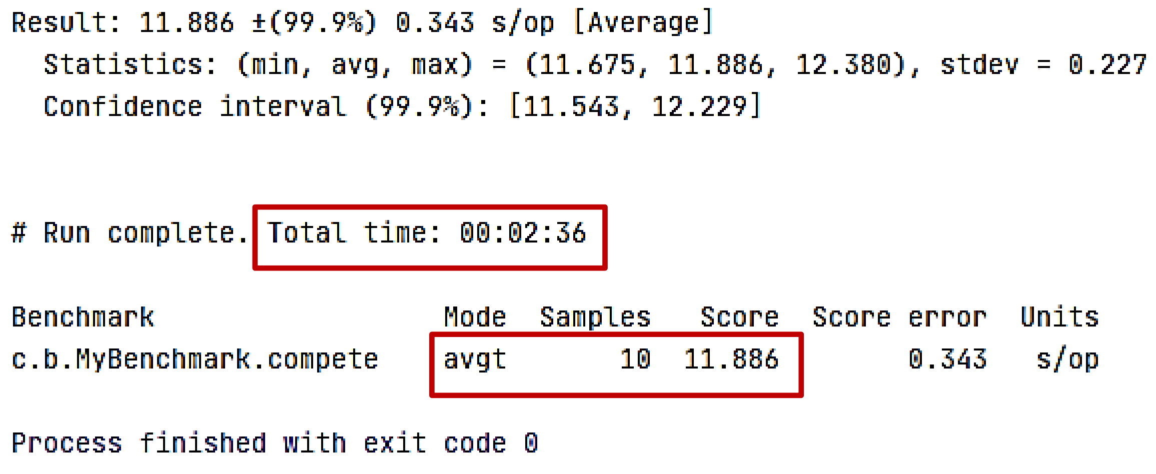


```
# Run progress: 0.00% complete, ETA 00:00:13
# Fork: 1 of 1
# Warmup Iteration   1: 12.244
12.384 s/op
# Warmup Iteration   2: 11.891
11.968 s/op
# Warmup Iteration   3: 11.706
11.757 s/op
Iteration   1: 11.729
11.793 s/op
Iteration   2: 11.699
11.728 s/op
Iteration   3: 11.897
11.930 s/op
Iteration   4: 12.329
12.380 s/op
```



```
Iteration   5: 11.75
11.815 s/op
Iteration   6: 11.61
11.677 s/op
Iteration   7: 11.901
11.961 s/op
Iteration   8: 11.642
11.675 s/op
Iteration   9: 12.115
12.145 s/op
Iteration  10: 11.7
11.753 s/op
```

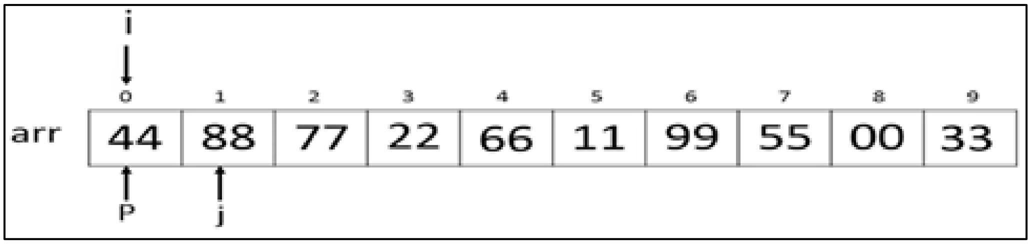
Avg(*Collection.sort*)-Avg (this algo) (s)	Total time(*Collection.sort*)-Total time(this algo) (s)
2.791 s	**35 s**

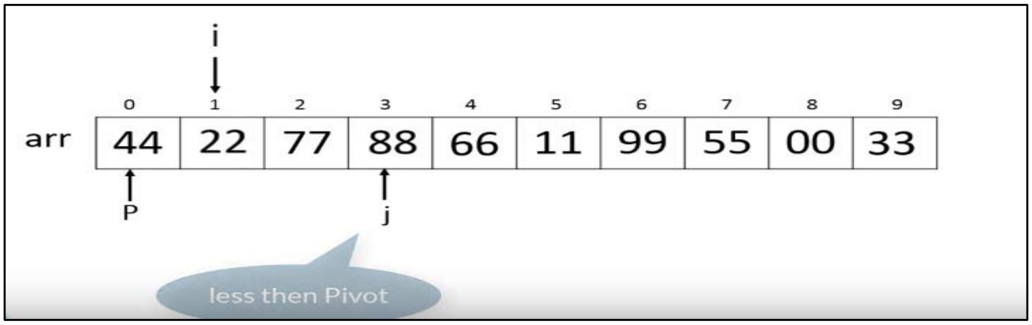


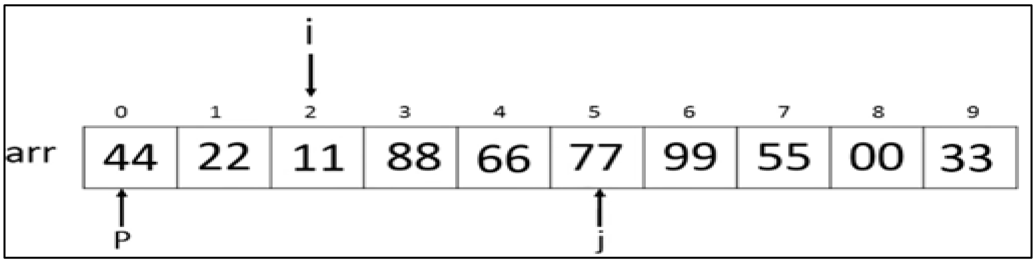
```
Result: 11.886 ±(99.9%) 0.343 s/op [Average]
  Statistics: (min, avg, max) = (11.675, 11.886, 12.380), stdev = 0.227
  Confidence interval (99.9%): [11.543, 12.229]


# Run complete. Total time: 00:02:36

Benchmark                      Mode  Samples   Score  Score error  Units
c.b.MyBenchmark.compete        avgt       10  11.886        0.343   s/op

Process finished with exit code 0
```

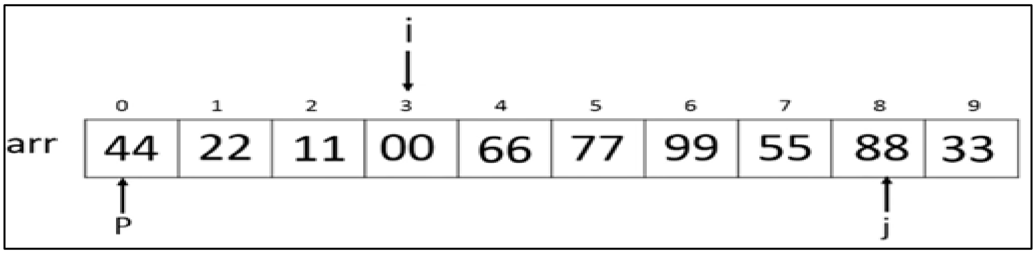
Hoare partition scheme quick sort

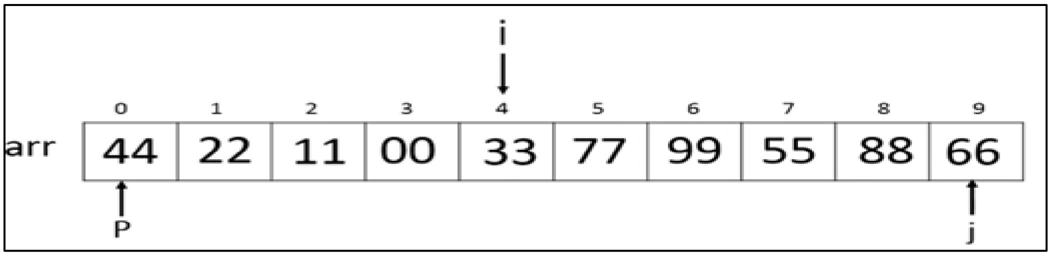
uses two indices that start at the ends of the array being partitioned, then move toward each other, until they detect an inversion: a pair of elements, one greater than or equal to the pivot, one less than or equal, that are in the wrong order relative to each other. The inverted elements are then swapped When the indices meet, the algorithm stops and returns the final index , Hoare's scheme is more efficient than Lomuto's partition scheme because it does three times fewer swaps on average , and it creates efficient partitions even when all values are equal. Like Lomuto's partition scheme, Hoare's partitioning also would cause Quicksort to degrade to $O(n^2)$ for already sorted input, if the pivot was chosen as the first or the last element.

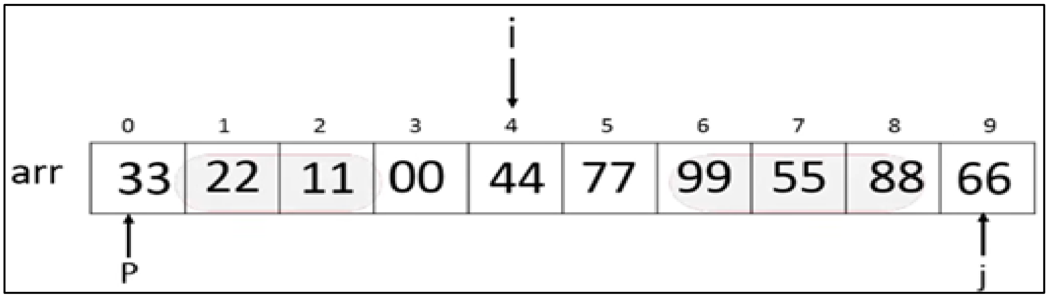


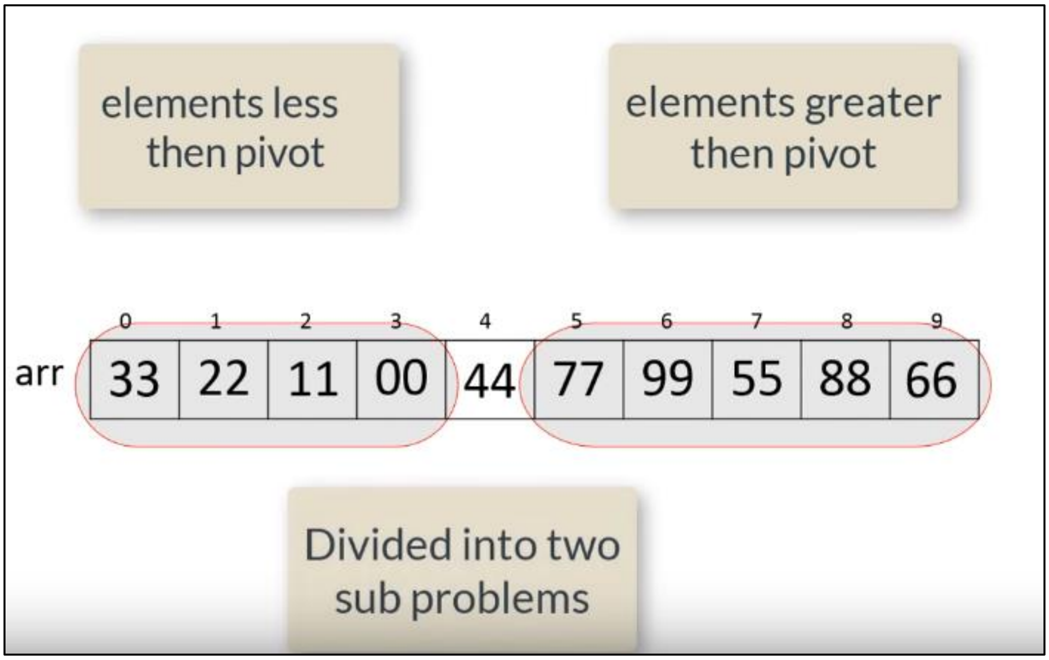






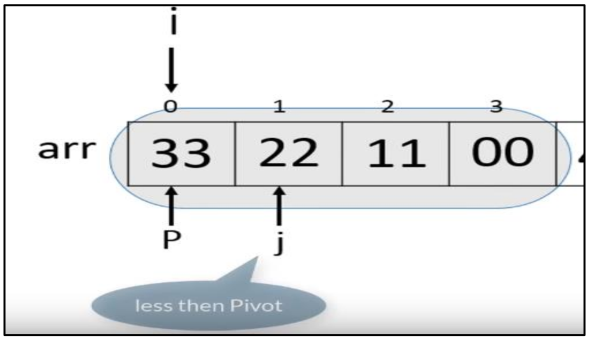





elements less then pivot

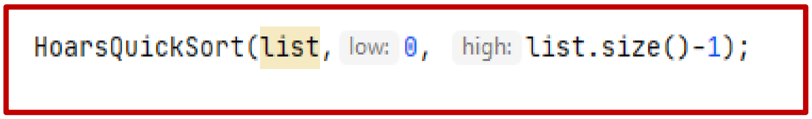
elements greater then pivot

Divided into two sub problems



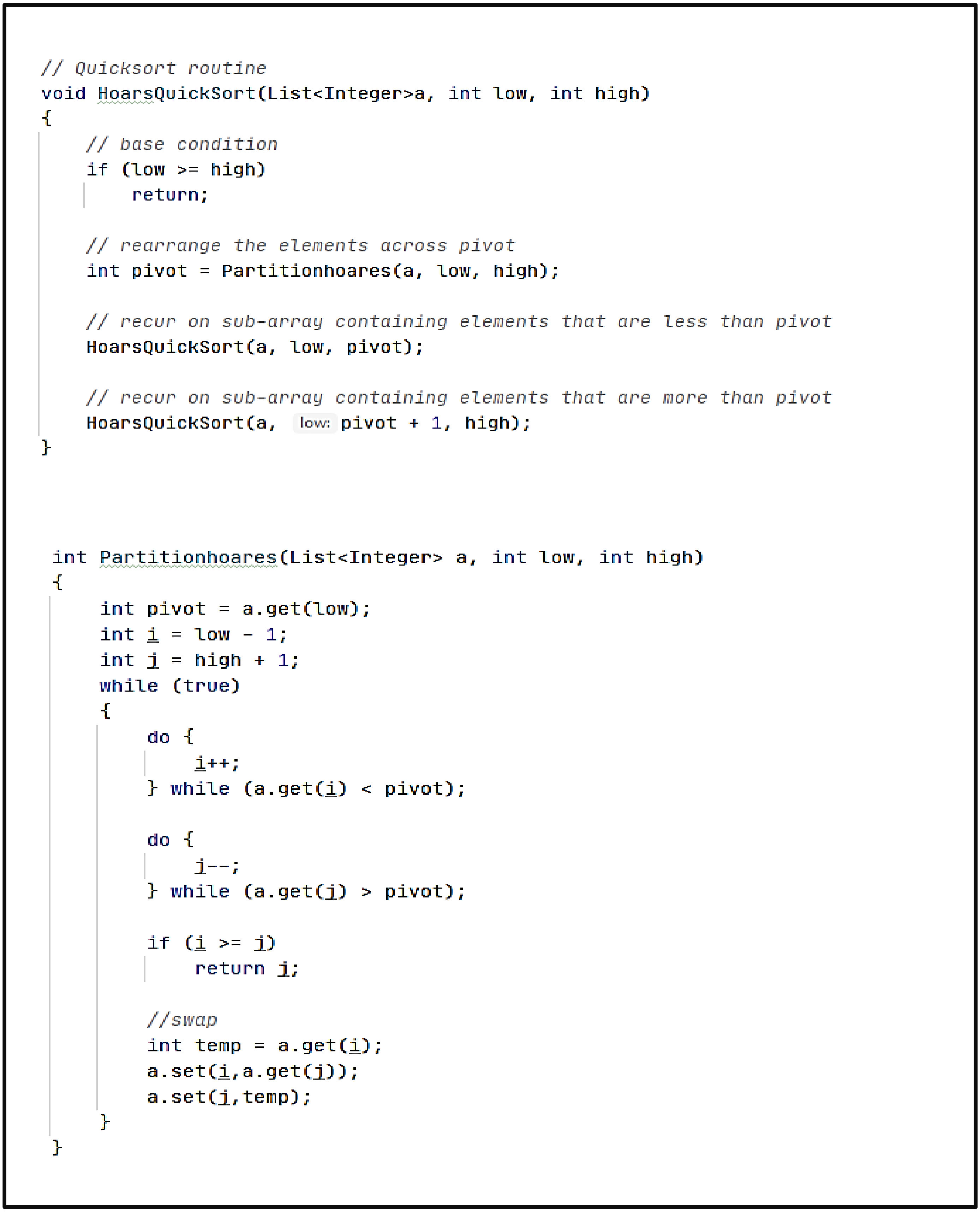
Implementation :

Inside sort(List list)method =>



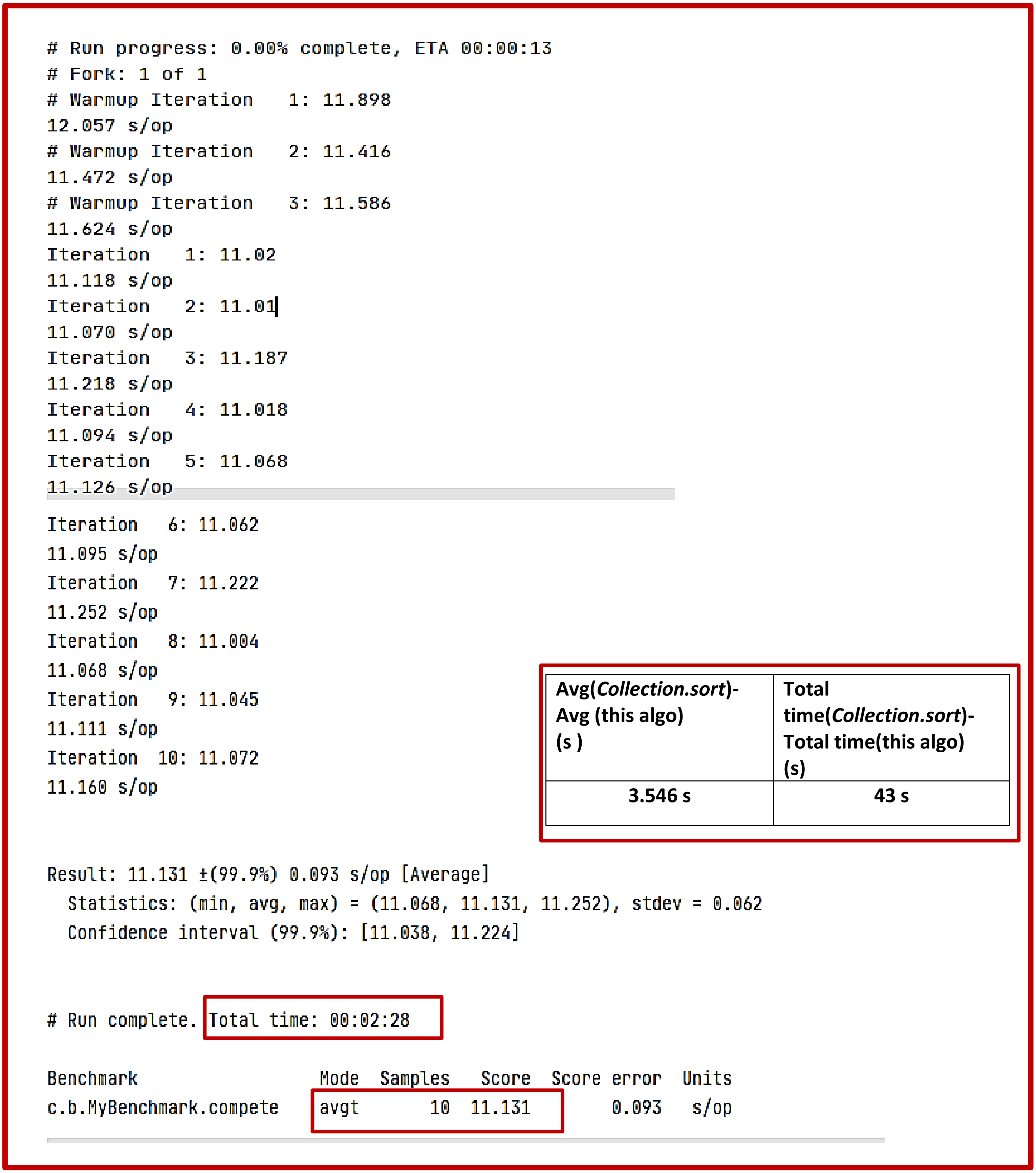
```
HoarsQuickSort(list, low: 0, high: list.size()-1);
```

Functions :



```
// Quicksort routine
void HoarsQuickSort(List<Integer>a, int low, int high)
{
    // base condition
    if (low >= high)
        return;

    // rearrange the elements across pivot
    int pivot = Partitionhoares(a, low, high);

    // recur on sub-array containing elements that are less than pivot
    HoarsQuickSort(a, low, pivot);

    // recur on sub-array containing elements that are more than pivot
    HoarsQuickSort(a, low: pivot + 1, high);
}



 int Partitionhoares(List<Integer> a, int low, int high)
 {
     int pivot = a.get(low);
     int i = low - 1;
     int j = high + 1;
     while (true)
     {
         do {
             i++;
         } while (a.get(i) < pivot);

         do {
             j--;
         } while (a.get(j) > pivot);

         if (i >= j)
             return j;

         //swap
         int temp = a.get(i);
         a.set(i,a.get(j));
         a.set(j,temp);
     }
 }
```

Results :



```
# Run progress: 0.00% complete, ETA 00:00:13
# Fork: 1 of 1
# Warmup Iteration   1: 11.898
12.057 s/op
# Warmup Iteration   2: 11.416
11.472 s/op
# Warmup Iteration   3: 11.586
11.624 s/op
Iteration   1: 11.02
11.118 s/op
Iteration   2: 11.01
11.070 s/op
Iteration   3: 11.187
11.218 s/op
Iteration   4: 11.018
11.094 s/op
Iteration   5: 11.068
11.126 s/op

Iteration   6: 11.062
11.095 s/op
Iteration   7: 11.222
11.252 s/op
Iteration   8: 11.004
11.068 s/op
Iteration   9: 11.045
11.111 s/op
Iteration  10: 11.072
11.160 s/op


Result: 11.131 ±(99.9%) 0.093 s/op [Average]
  Statistics: (min, avg, max) = (11.068, 11.131, 11.252), stdev = 0.062
  Confidence interval (99.9%): [11.038, 11.224]


# Run complete. Total time: 00:02:28

Benchmark                    Mode  Samples   Score  Score error  Units
c.b.MyBenchmark.compete      avgt       10  11.131        0.093   s/op
```

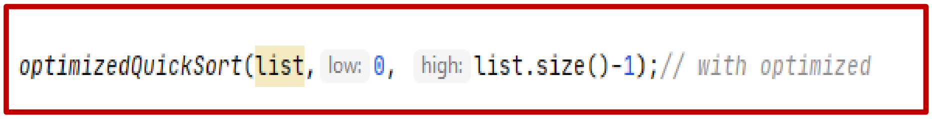
Avg(*Collection.sort*)-Avg (this algo) (s)	**Total time(*Collection.sort*)-Total time(this algo) (s)**
3.546 s	**43 s**

Optimized Quick sort (tail recursive Quick sort)

When the number of elements is below some threshold (perhaps ten elements), switch to a non-recursive sorting algorithm such as insertion sort that performs fewer swaps, comparisons or other operations on such small arrays. An insertion sort is a simple sort which is most effective on small list. It is quite slow at larger lists, but very fast with small lists

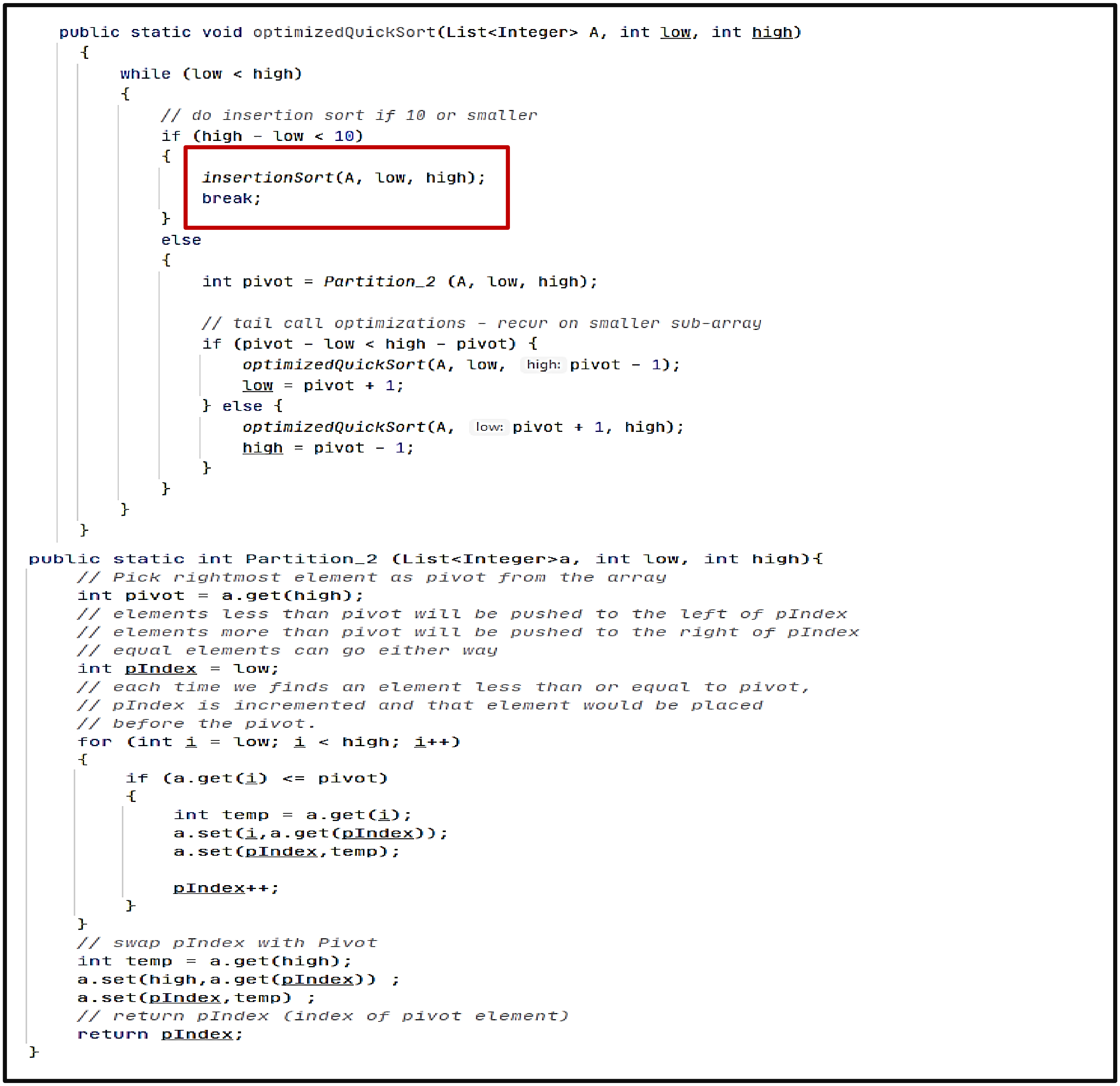
Implementation :

Inside sort(List list)method =>



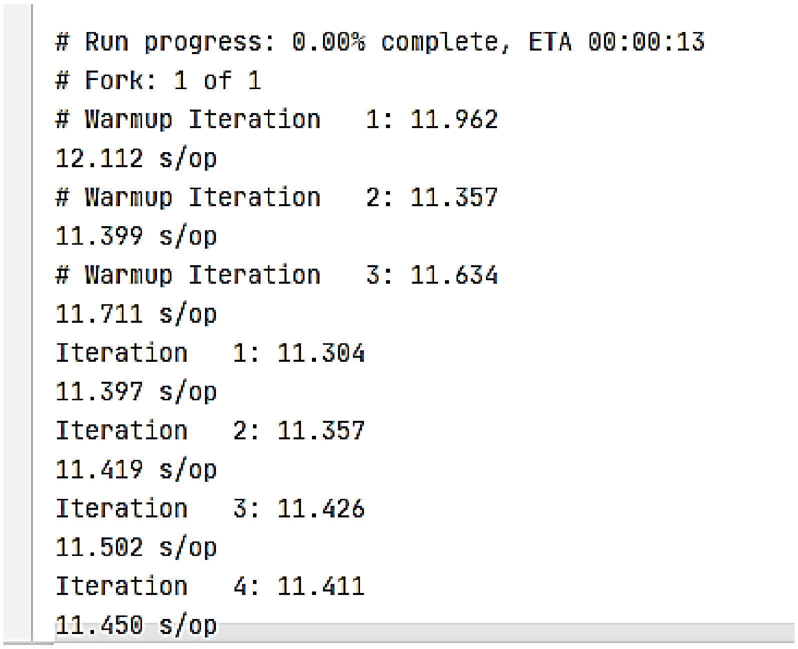
```
optimizedQuickSort(list, low: 0, high: list.size()-1);// with optimized
```

Functions :

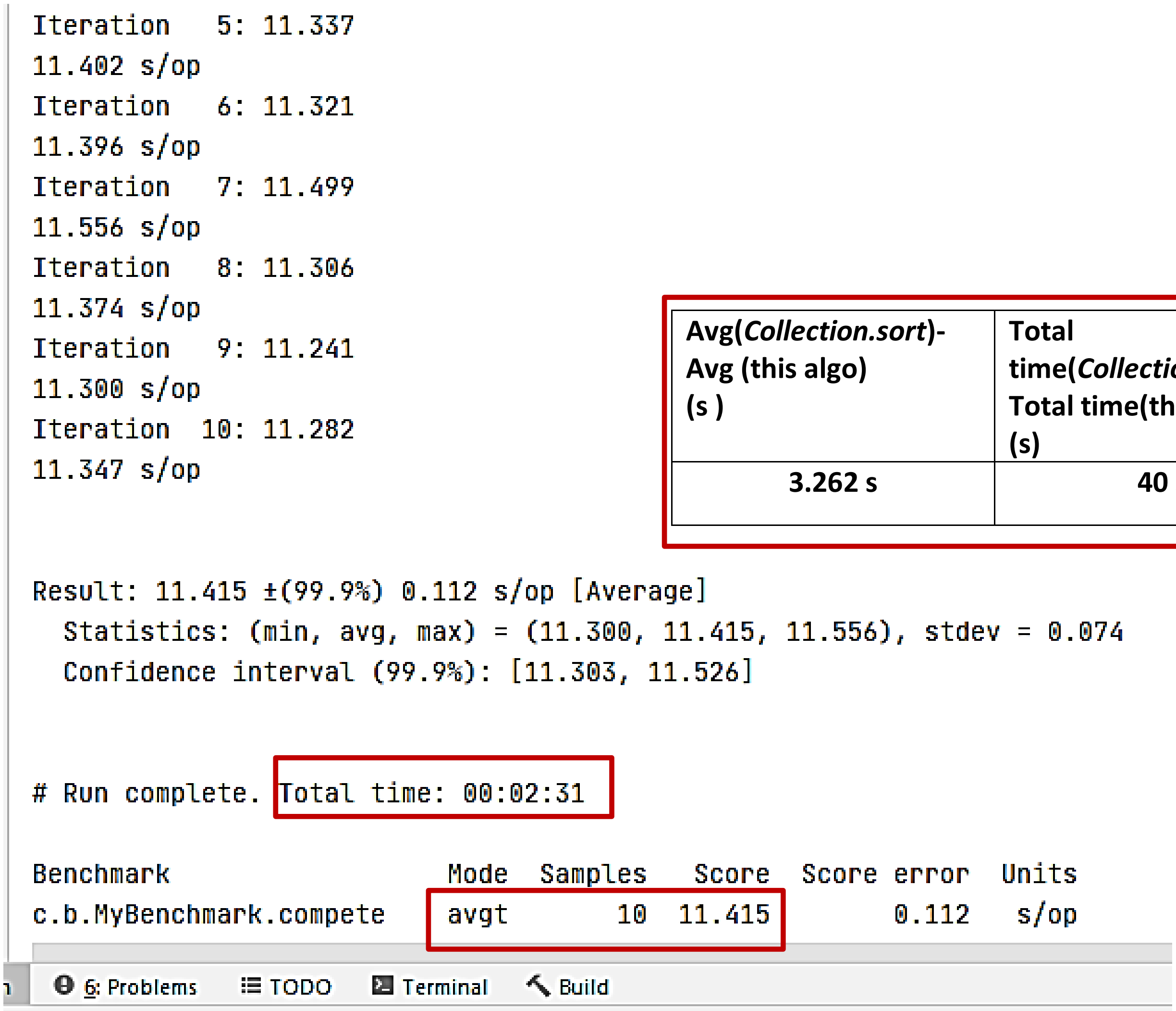


```
    public static void optimizedQuickSort(List<Integer> A, int low, int high)
    {
        while (low < high)
        {
            // do insertion sort if 10 or smaller
            if (high - low < 10)
            {
                insertionSort(A, low, high);
                break;
            }
            else
            {
                int pivot = Partition_2 (A, low, high);

                // tail call optimizations - recur on smaller sub-array
                if (pivot - low < high - pivot) {
                    optimizedQuickSort(A, low, high: pivot - 1);
                    low = pivot + 1;
                } else {
                    optimizedQuickSort(A, low: pivot + 1, high);
                    high = pivot - 1;
                }
            }
        }
    }
public static int Partition_2 (List<Integer>a, int low, int high){
    // Pick rightmost element as pivot from the array
    int pivot = a.get(high);
    // elements less than pivot will be pushed to the left of pIndex
    // elements more than pivot will be pushed to the right of pIndex
    // equal elements can go either way
    int pIndex = low;
    // each time we finds an element less than or equal to pivot,
    // pIndex is incremented and that element would be placed
    // before the pivot.
    for (int i = low; i < high; i++)
    {
        if (a.get(i) <= pivot)
        {
            int temp = a.get(i);
            a.set(i,a.get(pIndex));
            a.set(pIndex,temp);

            pIndex++;
        }
    }
    // swap pIndex with Pivot
    int temp = a.get(high);
    a.set(high,a.get(pIndex)) ;
    a.set(pIndex,temp) ;
    // return pIndex (index of pivot element)
    return pIndex;
}
```

Results :



```
# Run progress: 0.00% complete, ETA 00:00:13
# Fork: 1 of 1
# Warmup Iteration   1: 11.962
12.112 s/op
# Warmup Iteration   2: 11.357
11.399 s/op
# Warmup Iteration   3: 11.634
11.711 s/op
Iteration   1: 11.304
11.397 s/op
Iteration   2: 11.357
11.419 s/op
Iteration   3: 11.426
11.502 s/op
Iteration   4: 11.411
11.450 s/op
```



```
Iteration   5: 11.337
11.402 s/op
Iteration   6: 11.321
11.396 s/op
Iteration   7: 11.499
11.556 s/op
Iteration   8: 11.306
11.374 s/op
Iteration   9: 11.241
11.300 s/op
Iteration  10: 11.282
11.347 s/op
```

Avg(*Collection.sort*)-Avg (this algo) (s)	Total time(*Collection.sort*)-Total time(this algo) (s)
3.262 s	**40 s**

```
Result: 11.415 ±(99.9%) 0.112 s/op [Average]
  Statistics: (min, avg, max) = (11.300, 11.415, 11.556), stdev = 0.074
  Confidence interval (99.9%): [11.303, 11.526]


# Run complete. Total time: 00:02:31

Benchmark                     Mode  Samples   Score  Score error  Units
c.b.MyBenchmark.compete       avgt       10  11.415        0.112   s/op
```

Results compared to Collection.sort( ) .

Sort Algorithm	Result/Average iteration time s/op	Total time (minutes)	Confidence interval	(min,avg,max)
Collection.sort() :TimSort	14.677 s	3:11 m	[13.683,15.670]	(13.287,14.677,15.379)

Sort Algorithm	Result/Average iteration time s/op	Total time (minutes)	Confidence interval	(min,avg,max)	Avg(*Collection.sort*)-Avg (this algo) (s)	Total time(*Collection.sort*)-Total time(this algo)
Hoare quick sort	11.131 s	2:28 m	[11.038,11.224]	(11.068,11.131,11.252)	3.546 s	43 s
Optimized Quick sort	11.415 s	2:31 m	[11.303,11.526]	(11.300,11.415,11.556)	3.262 s	40 s
Tail Recursive Quick Sort .	11.886 s	2:36 m	[11.543,12.229]	(11.675,11.886,12.380)	2.791 s	35 s
Lomuto quick sort	13.247 s	2:55 m	[12.803,13.691]	(12.952,13.247,14.038)	1.43 s	16 s
Randomized Quick Sort	13.405 s	2:58 m	[12.939,13.870]	(13.153,13.405,14.201)	1.272 s	13 s
Dual pivot Quicksort	14.416 s	3:13 m	[13.473,15.359]	(13.531,14.416,15.430)	0.261 s	-2 s
Merge sort	17.156 s	3:45 m	[16.893,17.419]	(16.772,17.156,17.362)	-2.479 s	-43 s
Heap Sort	63.538 s	13:15 m	[54.844,72.233]	(58.942,63.538,78.794)	-48.861 s	10.06 minutes
Selection sort	Very long time	Very long time	Very long time	Very long time	-	-
Insertion sort	Very long time	Very long time	Very long time	Very long time	-	-
Bubble sort	Very long time	Very long time	Very long time	Very long time	-	-