**DHA Suffa University**



**Department of Computer Science**

**CS 201L – Data Structures and Algorithms Lab**

**Fall 2019**

**Lab 03 – Sorting Algorithms-I**

**Objective:**

Implementation of the following Sorting Algorithm

* Bubble Sort
* Selection Sort
* Insertion Sort
* Merge Sort

**Bubble Sort:**

Bubble sort is a simple sorting algorithm. This sorting algorithm is comparison-based algorithm in which each pair of adjacent elements is compared and the elements are swapped if they are not in order.

**How Bubble Sort Works?**

We take an unsorted array for our example. Bubble sort takes Ο(n2) time so we're keeping it short and precise.

Bubble Sort

Bubble sort starts with very first two elements, comparing them to check which one is greater.

Bubble Sort

In this case, value 33 is greater than 14, so it is already in sorted locations. Next, we compare 33 with 27.

Bubble Sort

We find that 27 is smaller than 33 and these two values must be swapped.

Bubble Sort

The new array should look like this −

Bubble Sort

Next we compare 33 and 35. We find that both are in already sorted positions.

Bubble Sort

Then we move to the next two values, 35 and 10.

Bubble Sort

We know then that 10 is smaller 35. Hence they are not sorted.

Bubble Sort

We swap these values. We find that we have reached the end of the array. After one iteration, the array should look like this −

Bubble Sort

To be precise, we are now showing how an array should look like after each iteration. After the second iteration, it should look like this −

Bubble Sort

Notice that after each iteration, at least one value moves at the end.

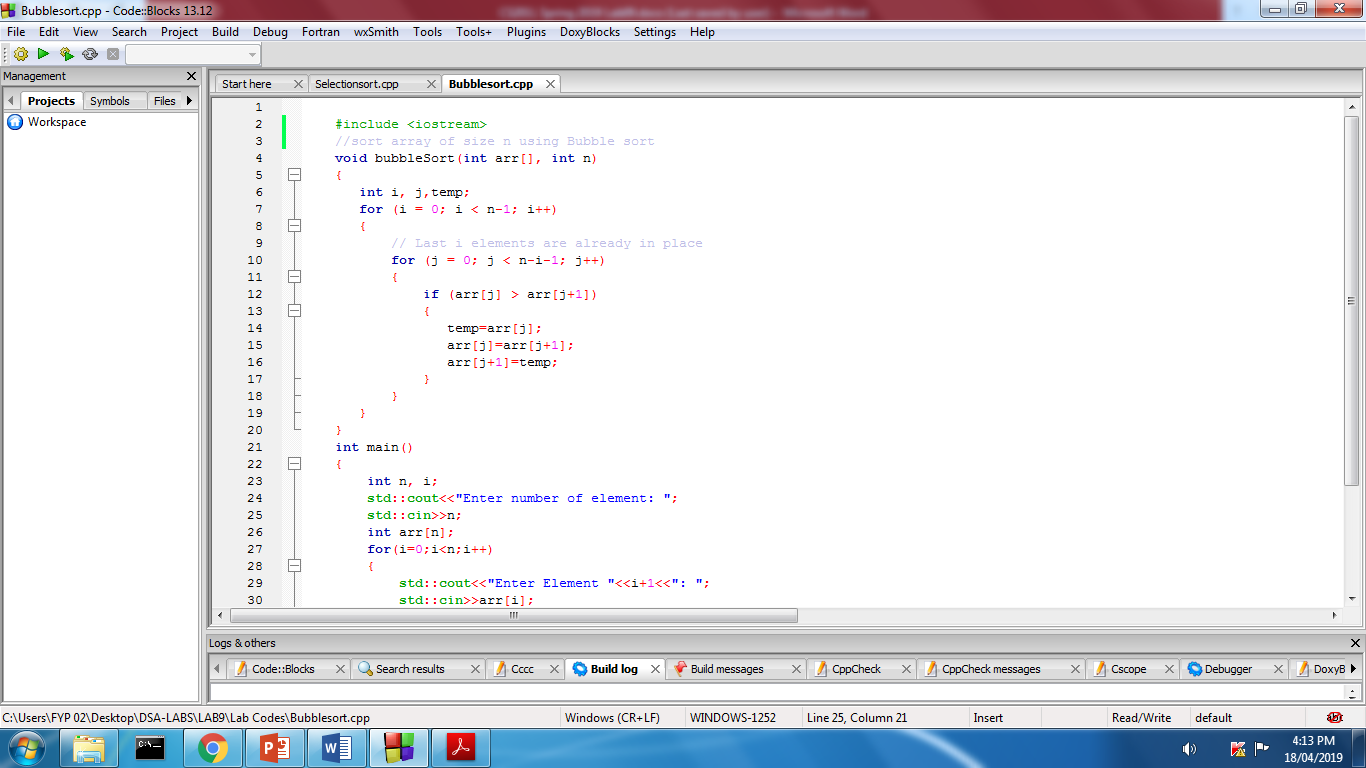
Bubble Sort

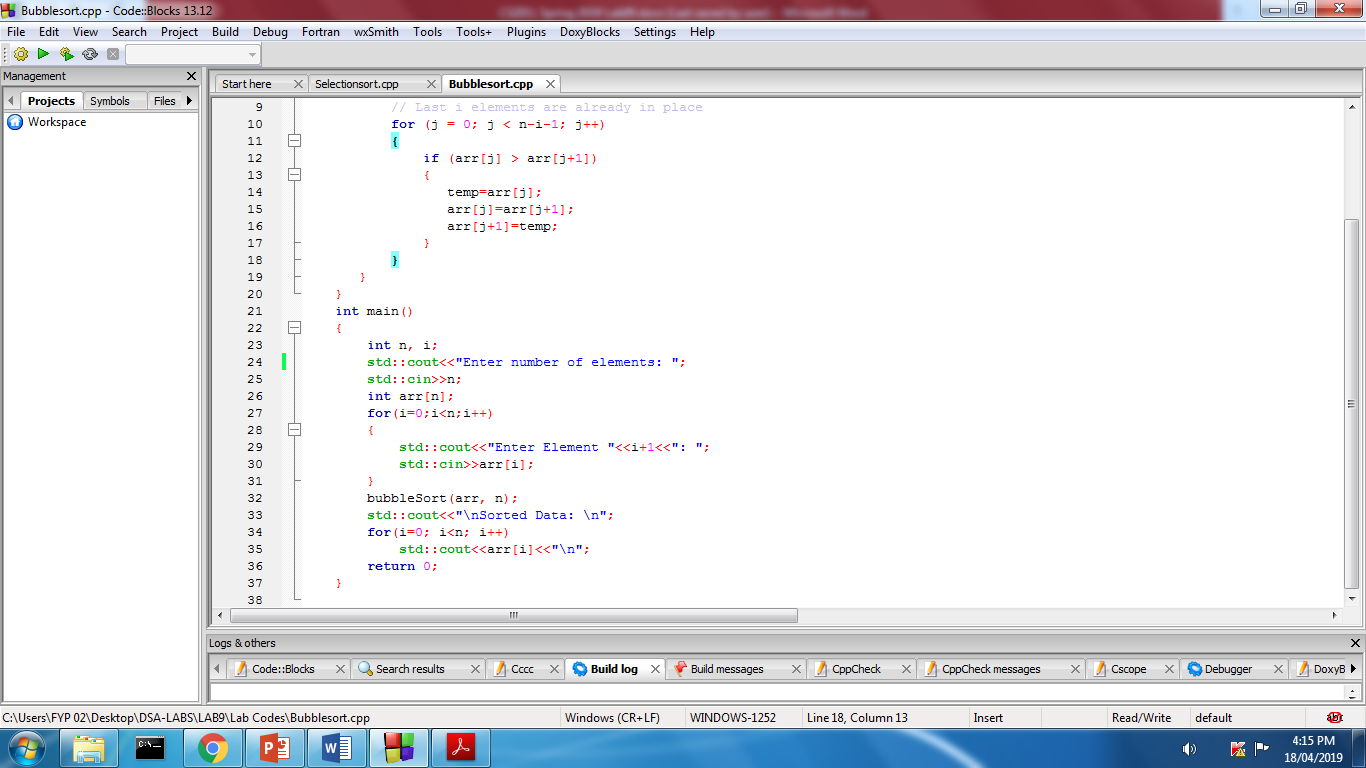
And when there's no swap required, bubble sorts learns that an array is completely sorted.

Bubble Sort

Now we should look into some practical aspects of bubble sort.

**Implementation:**





**Selection Sort:**

Selection sort is a simple sorting algorithm. This sorting algorithm is an in-place comparison-based algorithm in which the list is divided into two parts, the sorted part at the left end and the unsorted part at the right end. Initially, the sorted part is empty and the unsorted part is the entire list.

The smallest element is selected from the unsorted array and swapped with the leftmost element, and that element becomes a part of the sorted array. This process continues moving unsorted array boundary by one element to the right.

**How Selection Sort Works?**

Consider the following depicted array as an example.

Unsorted Array

For the first position in the sorted list, the whole list is scanned sequentially. The first position where 14 is stored presently, we search the whole list and find that 10 is the lowest value.

Selection Sort

So we replace 14 with 10. After one iteration 10, which happens to be the minimum value in the list, appears in the first position of the sorted list.

Selection Sort

For the second position, where 33 is residing, we start scanning the rest of the list in a linear manner.

Selection Sort

We find that 14 is the second lowest value in the list and it should appear at the second place. We swap these values.

Selection Sort

After two iterations, two least values are positioned at the beginning in a sorted manner.

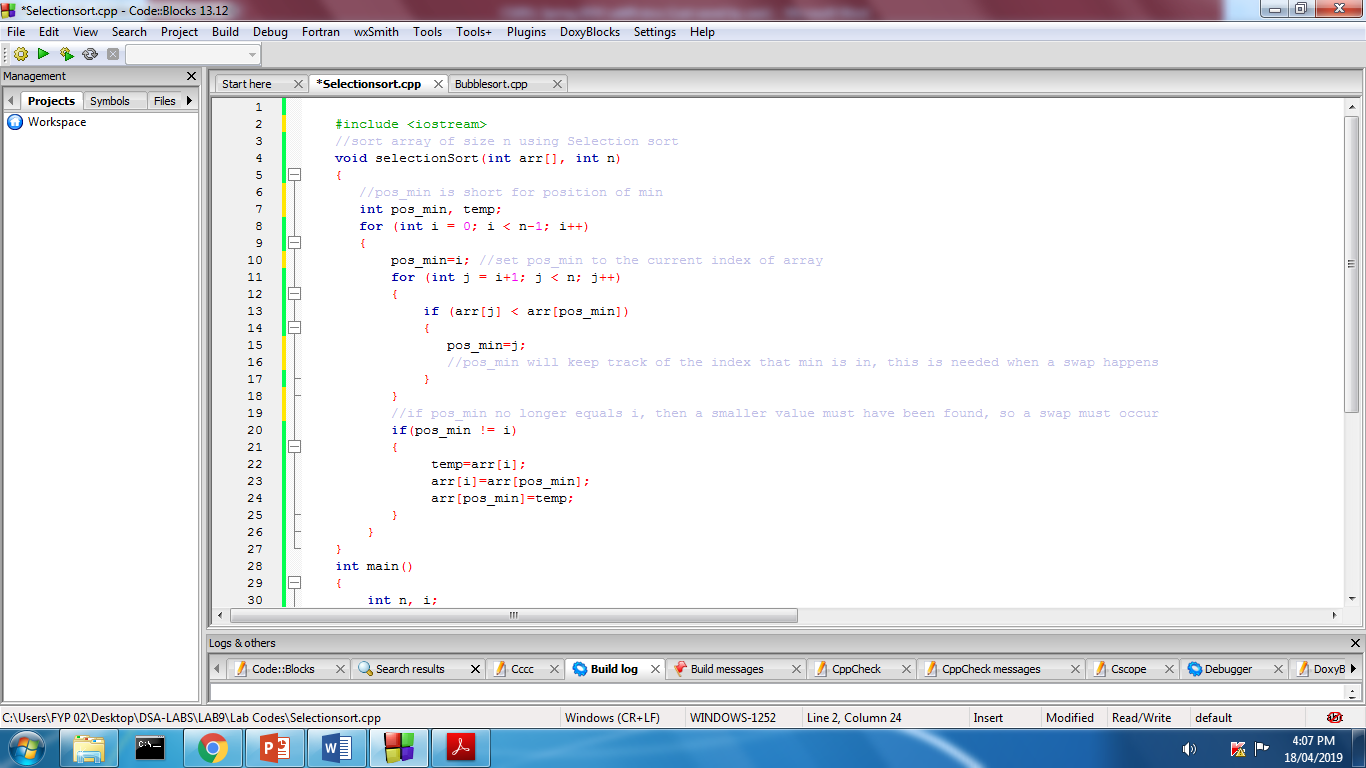
Selection Sort

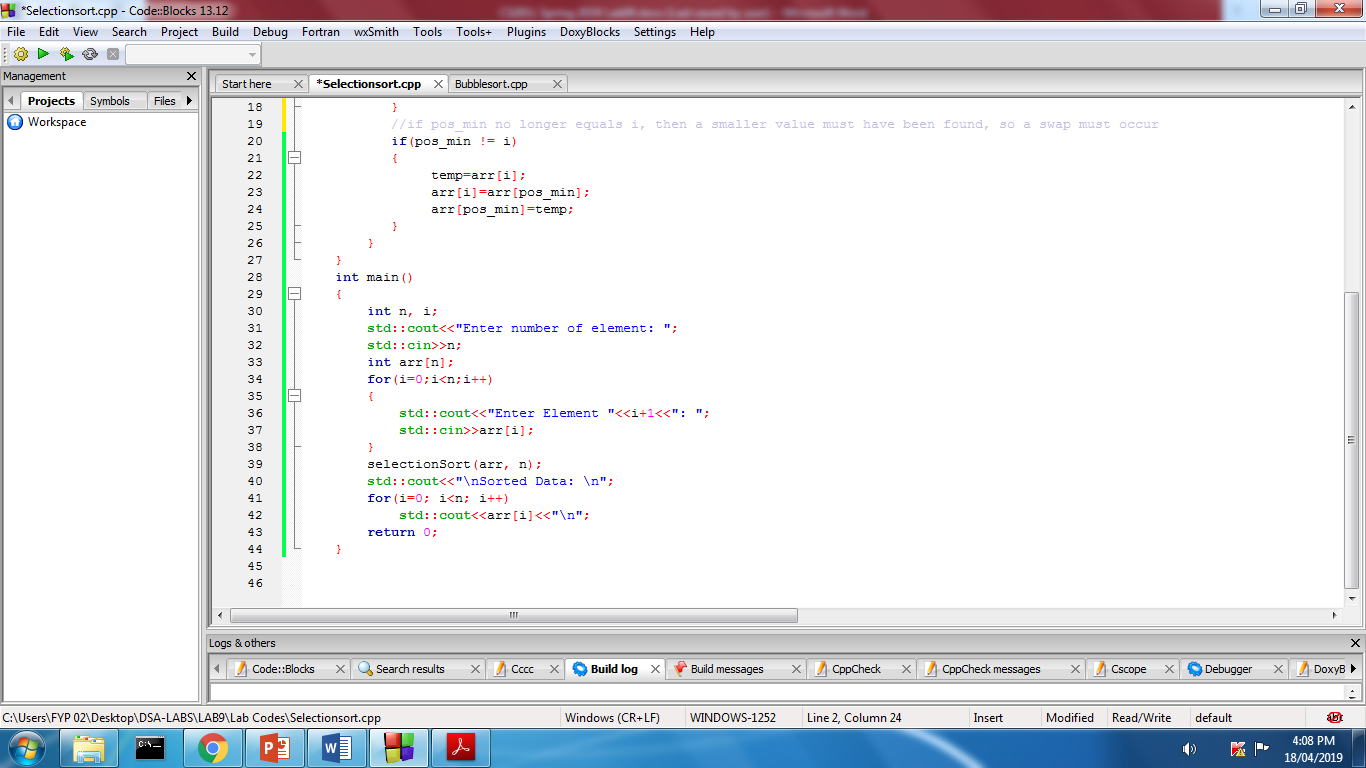
The same process is applied to the rest of the items in the array.

Following is a pictorial depiction of the entire sorting process −



**Implementation:**





**Insertion Sort:**

This is an in-place comparison-based sorting algorithm. Here, a sub-list is maintained which is always sorted. For example, the lower part of an array is maintained to be sorted. An element which is to be inserted in this sorted sub-list, has to find its appropriate place and then it has to be inserted there. Hence the name, insertion sort.

The array is searched sequentially and unsorted items are moved and inserted into the sorted sub-list (in the same array).

**How Insertion Sort Works?**

We take an unsorted array for our example.

Unsorted Array

Insertion sort compares the first two elements.

Insertion Sort

It finds that both 14 and 33 are already in ascending order. For now, 14 is in sorted sub-list.

Insertion Sort

Insertion sort moves ahead and compares 33 with 27.

Insertion Sort

And finds that 33 is not in the correct position.

Insertion Sort

It swaps 33 with 27. It also checks with all the elements of sorted sub-list. Here we see that the sorted sub-list has only one element 14, and 27 is greater than 14. Hence, the sorted sub-list remains sorted after swapping.

Insertion Sort

By now we have 14 and 27 in the sorted sub-list. Next, it compares 33 with 10.

Insertion Sort

These values are not in a sorted order.

Insertion Sort

So we swap them.

Insertion Sort

However, swapping makes 27 and 10 unsorted.

Insertion Sort

Hence, we swap them too.

Insertion Sort

Again we find 14 and 10 in an unsorted order.

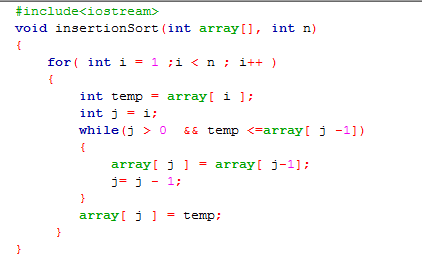
Insertion Sort

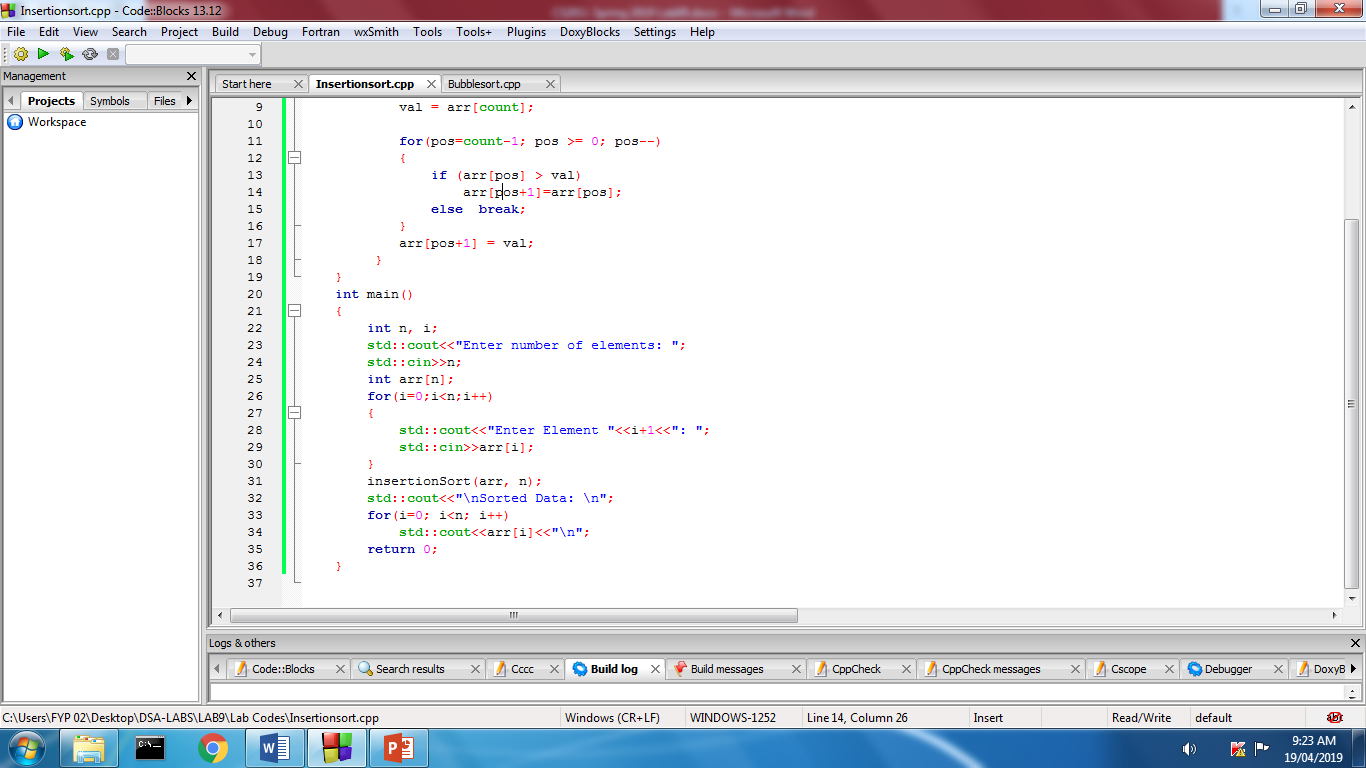
We swap them again. By the end of third iteration, we have a sorted sub-list of 4 items.

Insertion Sort

This process goes on until all the unsorted values are covered in a sorted sub-list.

**Implementation:**





**Merge Sort:**

Merge sort is a sorting technique based on divide and conquer technique. Merge sort first divides the array into equal halves and then combines them in a sorted manner.

**How Merge Sort Works?**

To understand merge sort, we take an unsorted array as the following −

Unsorted Array

We know that merge sort first divides the whole array iteratively into equal halves unless the atomic values are achieved. We see here that an array of 8 items is divided into two arrays of size 4.

Merge Sort Division

This does not change the sequence of appearance of items in the original. Now we divide these two arrays into halves.

Merge Sort Division

We further divide these arrays and we achieve atomic value which can no more be divided.

Merge Sort Division

Now, we combine them in exactly the same manner as they were broken down. Please note the color codes given to these lists.

We first compare the element for each list and then combine them into another list in a sorted manner. We see that 14 and 33 are in sorted positions. We compare 27 and 10 and in the target list of 2 values we put 10 first, followed by 27. We change the order of 19 and 35 whereas 42 and 44 are placed sequentially.

Merge Sort Combine

In the next iteration of the combining phase, we compare lists of two data values, and merge them into a list of found data values placing all in a sorted order.

Merge Sort Combine

After the final merging, the list should look like this −

Merge Sort

Now we should learn some programming aspects of merge sorting.

### **Algorithm**

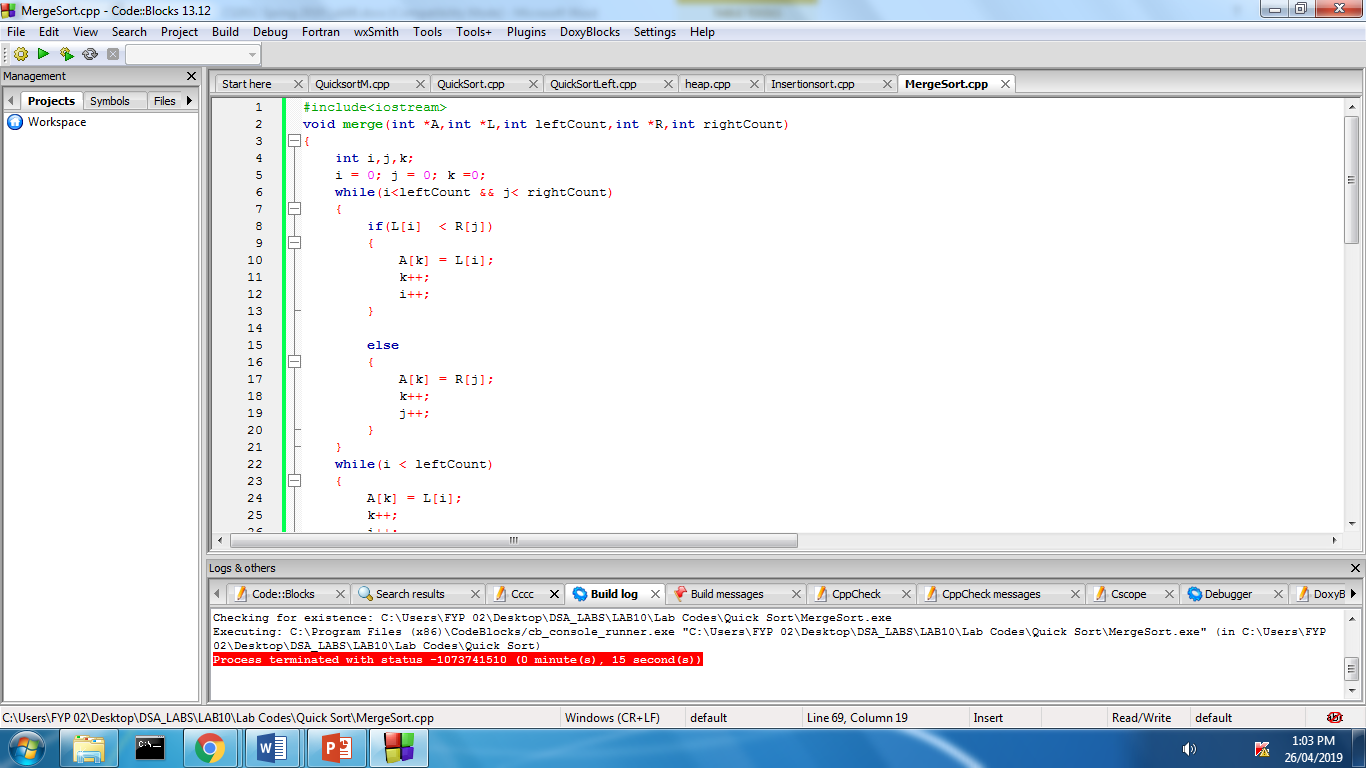
Merge sort keeps on dividing the list into equal halves until it can no more be divided. By definition, if it is only one element in the list, it is sorted. Then, merge sort combines the smaller sorted lists keeping the new list sorted too.

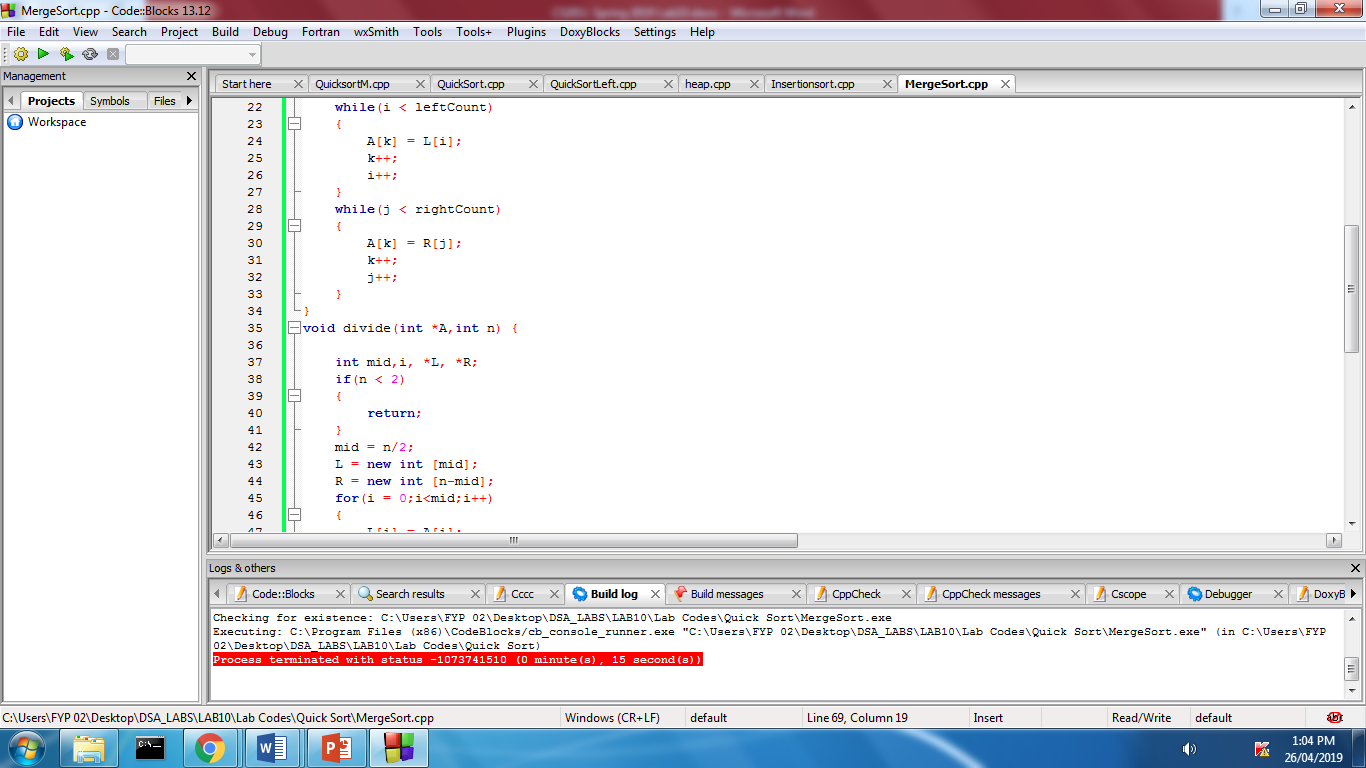
Step 1 − if it is only one element in the list it is already sorted, return.

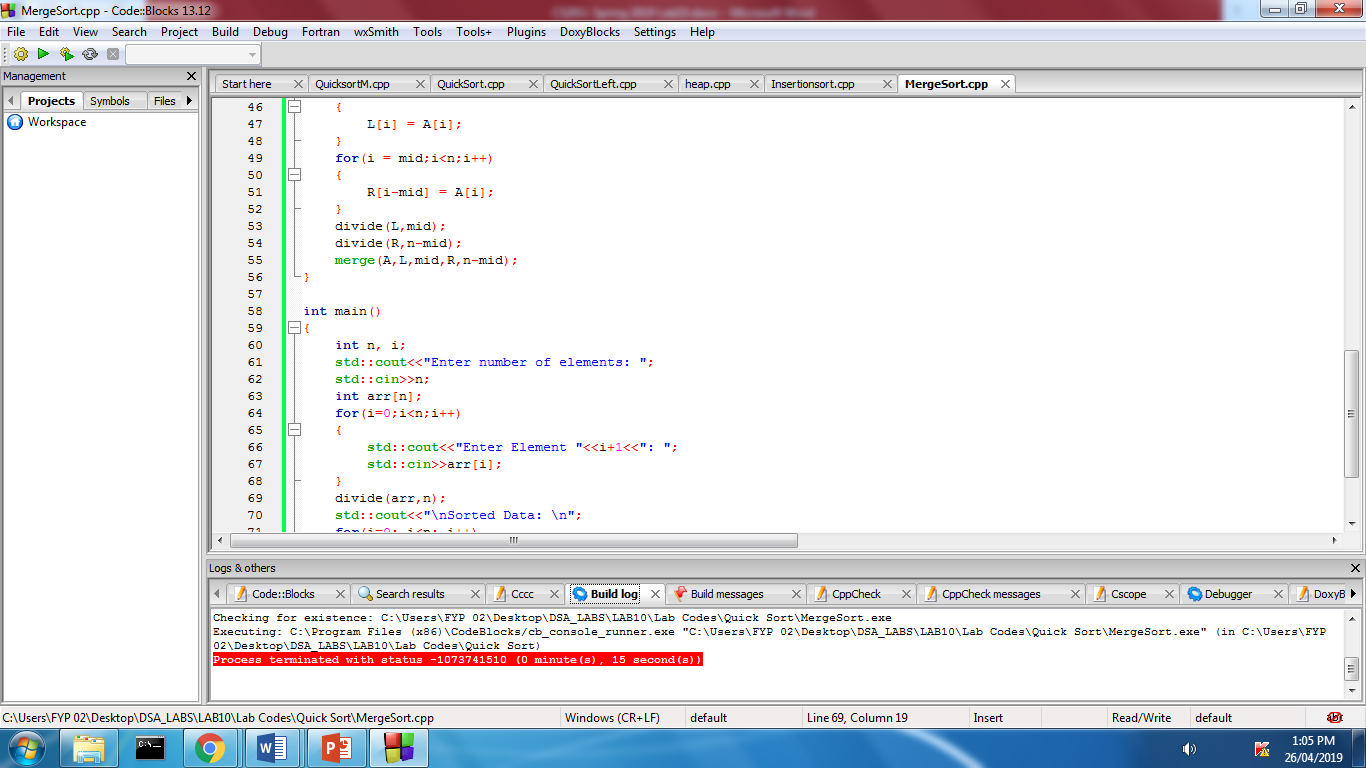
Step 2 − divide the list recursively into two halves until it can no more be divided.

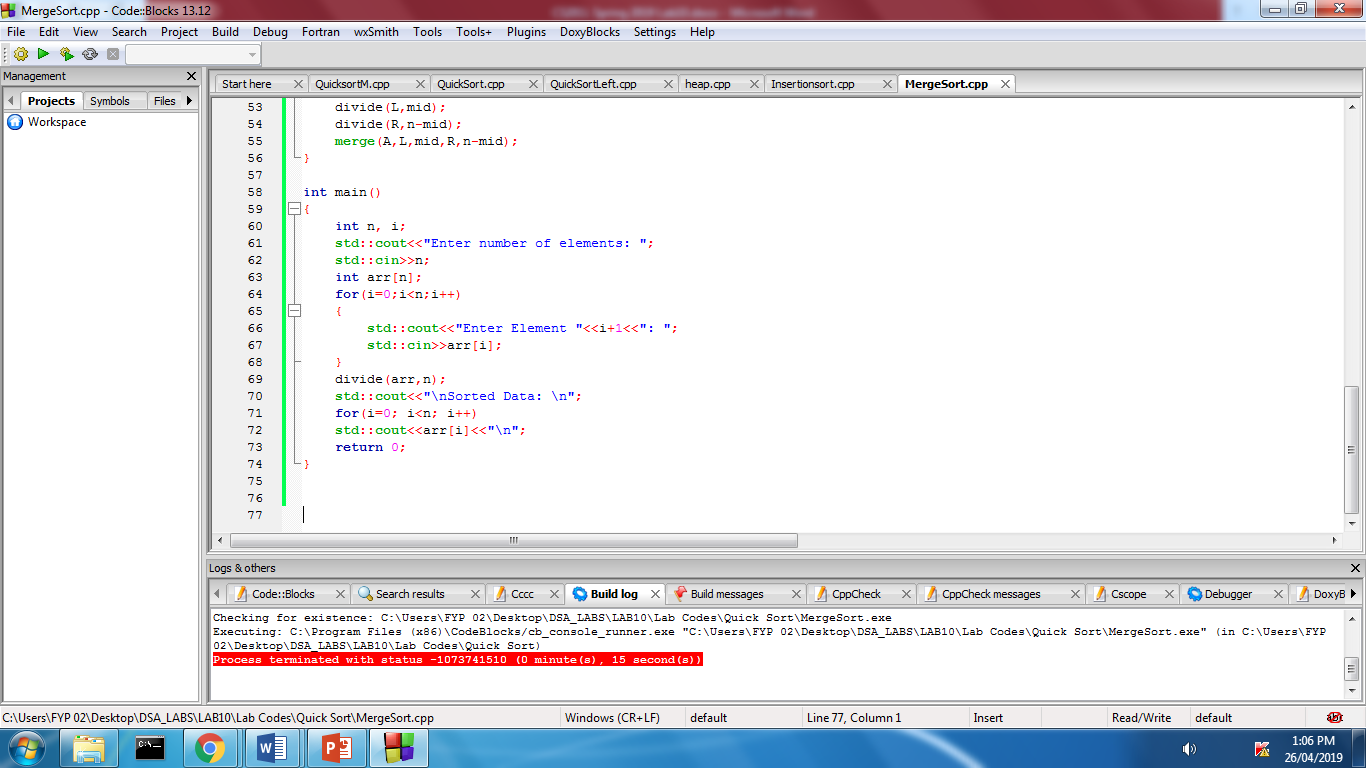
Step 3 − merge the smaller lists into new list in sorted order.

**Implementation:**









**Lab Task:**

Using the sorting algorithms discussed in lab, write a program that will sort the given names of employee from a predefined array of names of size N (you can assume any size).

**Assignment:**

Given an input file, write a program that will read the file in a multidimensional dynamic array of employee.

Sort the column employee name by applying each sorting algorithm in such a way that their respective data in the rows is shifted with the names too, while sorting.

# **Submission Guidelines**

* **Write C++ code , separate function for each operation.**
* **Place your file in a folder named with your rollNo (cs172xxx) where xxx is your 3 digit rollno.**
* **Upload it on LMS.**

**Note:**

**Remember -100 policy for plagiarism.**