Merge Sort

1. Mergesort guarantees to sort an array in _______ time, regardless of the input:

   D. Logarithmic time

2. The main disadvantage of MergeSort is:

   B. It uses extra space in proportion to the size of the input

3. Merge sort makes use of which common algorithm strategy?

   D. Divide and conquer

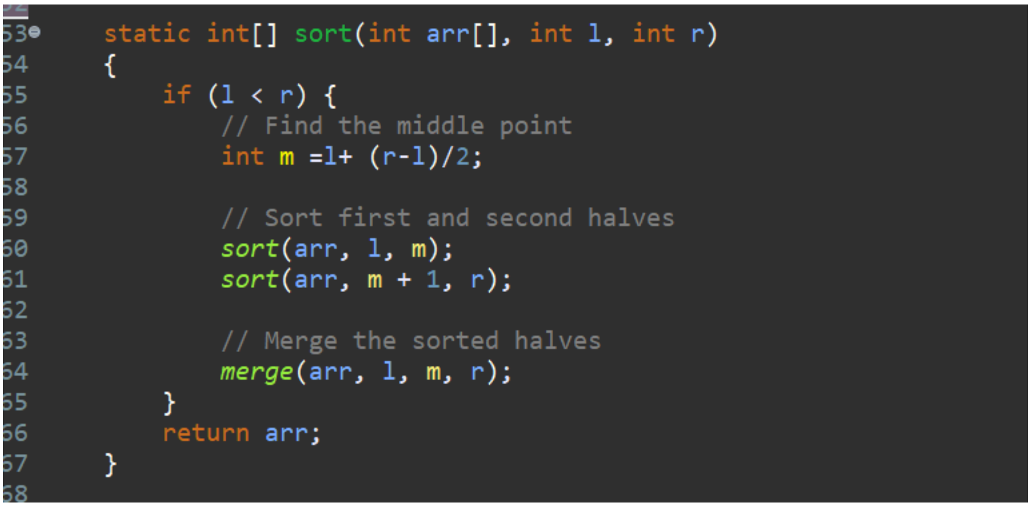
4. Which sorting algorithm will take the least time when all elements of the input array are identical?

   B. MergeSort

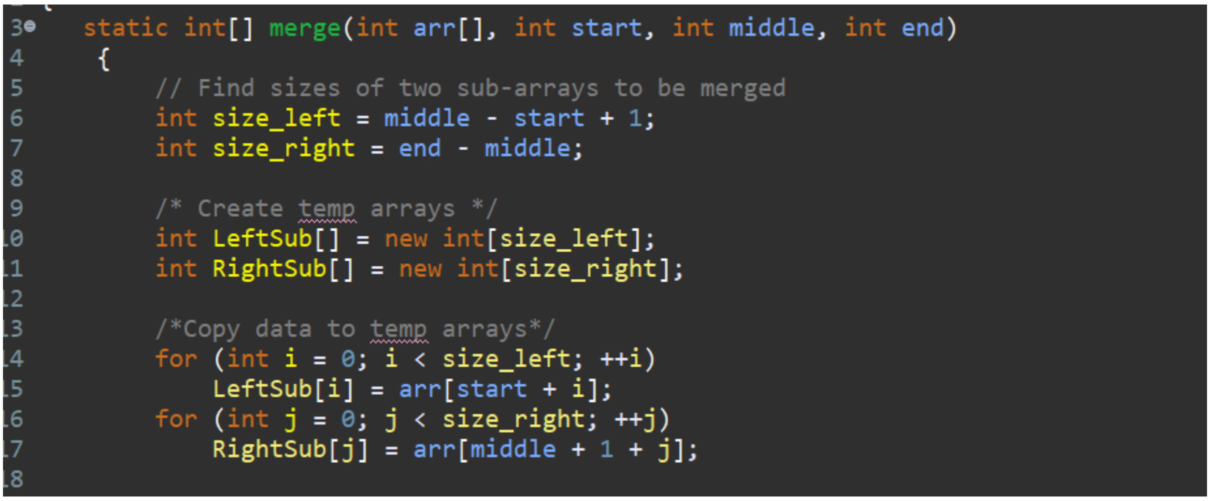
5. Which sorting algorithm should you use when the order of input is not known?

   A. Mergesort

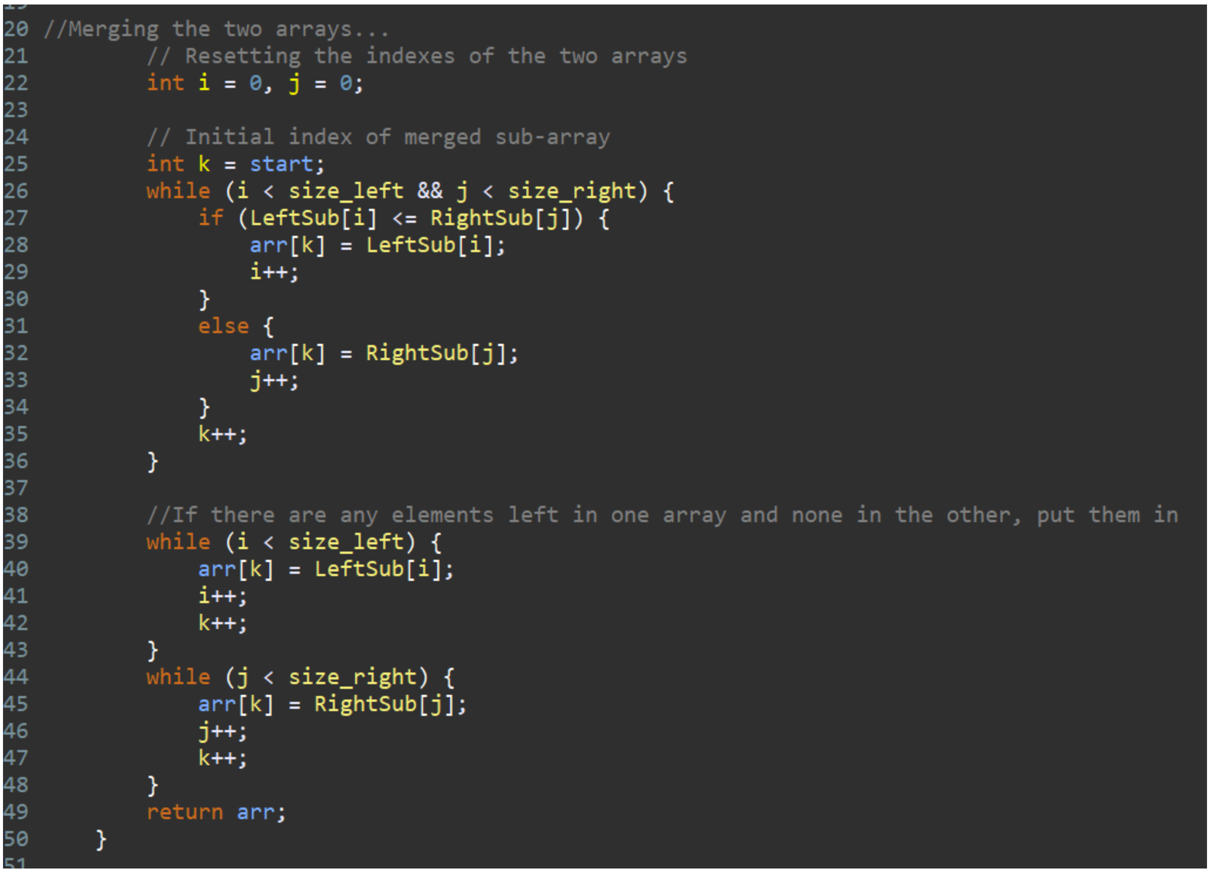
Implementation of Merge Sort



```
static int[] sort(int arr[], int l, int r)
{
    if (l < r) {
        // Find the middle point
        int m =l+ (r-l)/2;

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

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



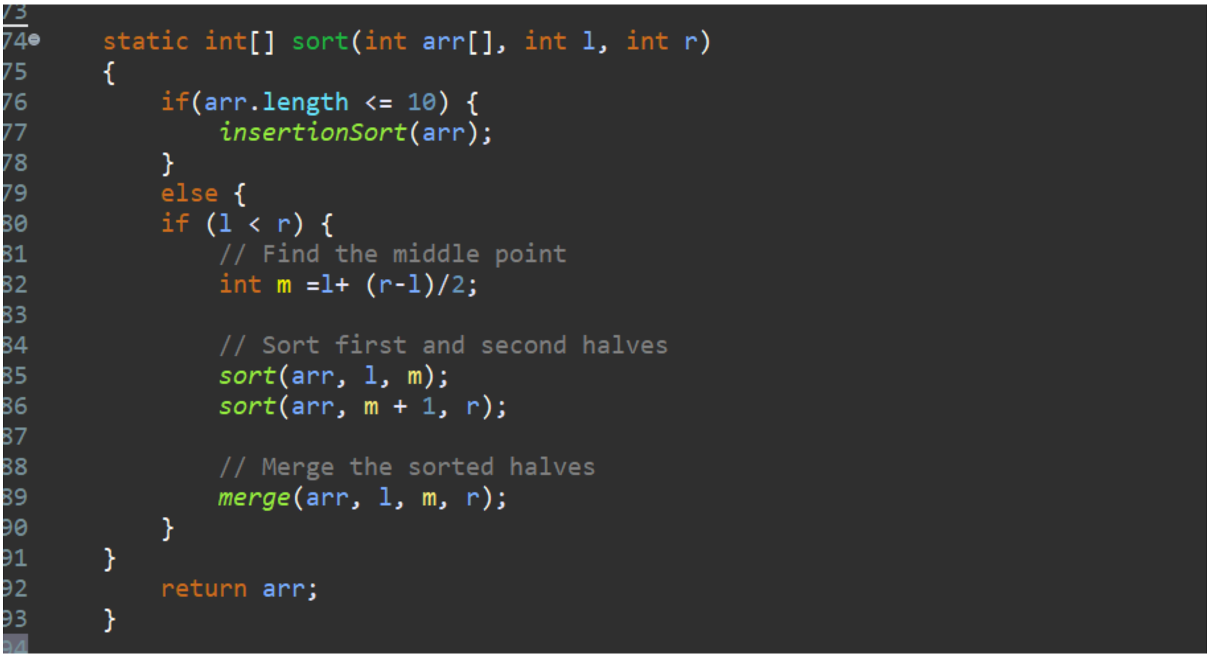
```
static int[] merge(int arr[], int start, int middle, int end)
 {
     // Find sizes of two sub-arrays to be merged
     int size_left = middle - start + 1;
     int size_right = end - middle;

     /* Create temp arrays */
     int LeftSub[] = new int[size_left];
     int RightSub[] = new int[size_right];

     /*Copy data to temp arrays*/
     for (int i = 0; i < size_left; ++i)
         LeftSub[i] = arr[start + i];
     for (int j = 0; j < size_right; ++j)
         RightSub[j] = arr[middle + 1 + j];
```



```
//Merging the two arrays...
        // Resetting the indexes of the two arrays
        int i = 0, j = 0;

        // Initial index of merged sub-array
        int k = start;
        while (i < size_left && j < size_right) {
            if (LeftSub[i] <= RightSub[j]) {
                arr[k] = LeftSub[i];
                i++;
            }
            else {
                arr[k] = RightSub[j];
                j++;
            }
            k++;
        }

        //If there are any elements left in one array and none in the other, put them in
        while (i < size_left) {
            arr[k] = LeftSub[i];
            i++;
            k++;
        }
        while (j < size_right) {
            arr[k] = RightSub[j];
            j++;
            k++;
        }
        return arr;
    }
```

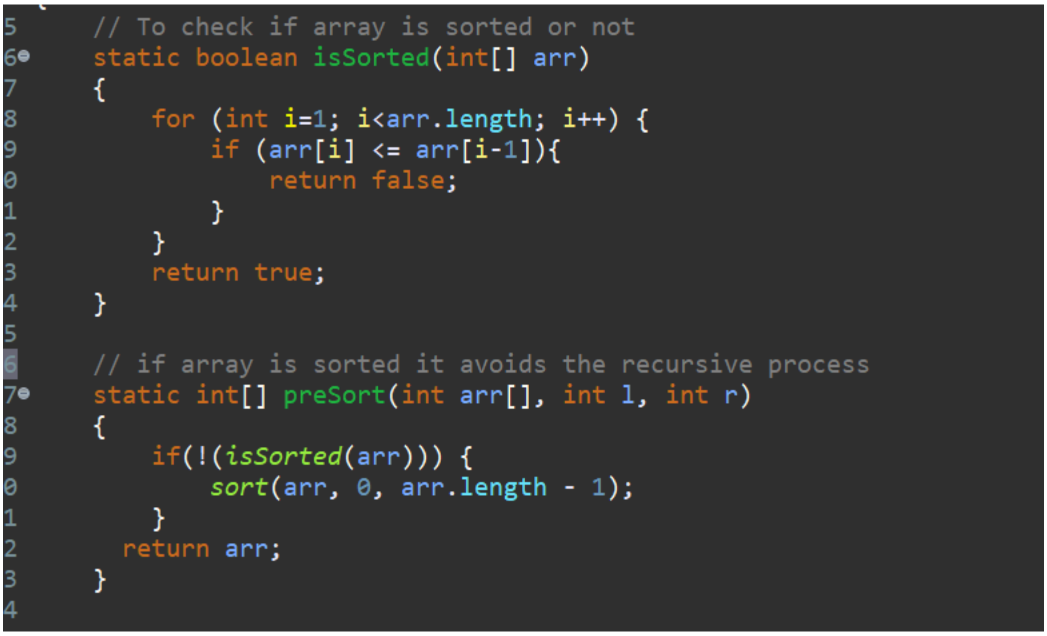
Enhanced version of Merge sort.

If there are 10 or less elements in the array, use insertion sort



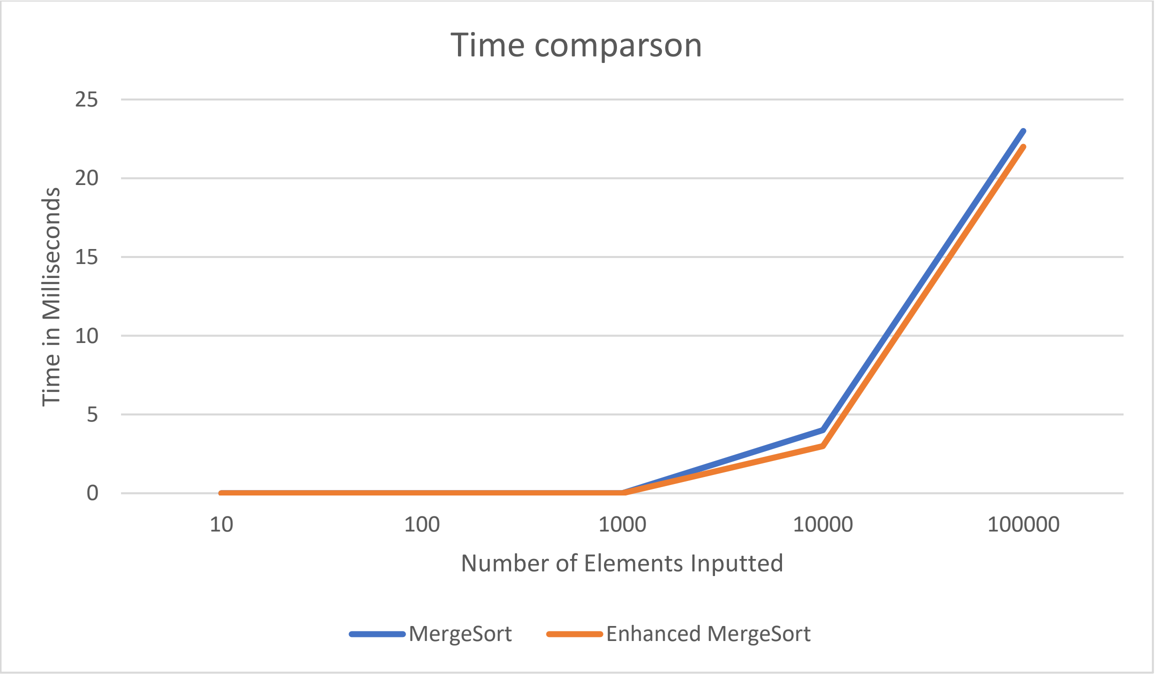
```
    static int[] sort(int arr[], int l, int r)
    {
        if(arr.length <= 10) {
            insertionSort(arr);
        }
        else {
        if (l < r) {
            // Find the middle point
            int m =l+ (r-l)/2;

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

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

I added a function that iterates through to see if the array is already sorted, in pre-sort, if it isn't sorted then it goes ahead and calls the recursive sort function.



```
    // To check if array is sorted or not
    static boolean isSorted(int[] arr)
    {
        for (int i=1; i<arr.length; i++) {
            if (arr[i] <= arr[i-1]){
                return false;
            }
        }
        return true;
    }

    // if array is sorted it avoids the recursive process
    static int[] preSort(int arr[], int l, int r)
    {
        if(!(isSorted(arr))) {
            sort(arr, 0, arr.length - 1);
        }
      return arr;
    }
```

	10	100	1000	10000	100000
MergeSort	0.0	0.0	0.0	4.0	23.0
Enhanced MergeSort	0.0	0.0	0.0	3.0	22.0


Time comparson
Time in Milliseconds
Number of Elements Inputted
MergeSort
Enhanced MergeSort