ID	Name	Email	Group
133722	Salma Mohamed Yassin	Salma133722@bue.edu.eg	A6
129128	Nouran Moataz Mosallam	Nouran129128@bue.edu.eg	A5

Table of Contents:

1. Implementation:

```cpp
// Salma Mohamed Yassin - 133722
// Nouran Moataz Mosallam - 129128

#include<iostream>
#include <ctime>
#include <thread>
#include <iomanip>

using namespace std;

int *Arr;                       // Initializing the dynamic array for the Merge Sort Function
int *Arr2;                      // Initializing a copy of the dynamic array for the Quick Sort Function
int size;                       // An integer holding the dynamic arrays' size

int counterMerge = 0;           // Counts the Merge Sort steps
int counterQuick = 0;           // Counts the Quick Sort steps

clock_t mergeStart;             // To holds the starting time of the merge sort function
clock_t quickStart;             // To holds the starting time of the quick sort function

long double mergeTotal;             // to Calculates the duration of the mergeSort function
long double quickTotal;             // to Calculates the duration of the quickSort function

void display(int *arr, int size, int sort){     /********************/
    if (sort == 1)                              /*    A function to  */
        cout << "\nMerge";                      /* display the array */
    if (sort == 2)                              /*      elements     */
        cout << "\nQuick";                      /********************/

    cout << "Sorted: ";
    for (int i = 0; i < size; i++){
        cout << arr[i] << " ";
    }

    cout << endl;

    if (sort == 1){
        cout << "Merge Sort Counter: " << counterMerge;     // Display the number of steps in mergeSort
    }
    else if (sort == 2){
        cout << "Quick Sort Counter: " << counterQuick;     // Display the number of steps in quickSort
    }
    cout << endl;
    cout << endl;

}

/*************/
/* Merge Sort */
/*************/

```

```
void merge(int *arr, int *arrLeft, int *arrRight, int sizeLeft, int sizeRight, int sizeArray){

    int i = 0;                                  // i to manage the index of the temporary left array
    int j = 0;                                  // j to manage the index of the temporary right array
    int k = 0;                                  // k to manage the index of the real array

    /***********************/
    /* Sorting and Merging */
    /***********************/

    while (i < sizeLeft && j < sizeRight) {     // Loop untill the end of the Right array or the Left array is reached
        counterMerge++;

        if (arrLeft[i] < arrRight[j])           // If the number in the left array is less then the number in the right
            arr[k++] = arrLeft[i++];            // put the number which was in the left array in the real array and increment both the real and the left arrays' indexes

        else                                    // If the number in the right array is less then the number in the left
            arr[k++] = arrRight[j++];           // put the number which was in the right array in the real array and increment both the real and the right arrays' indexes
    }

    while (i < sizeLeft) {                      // If the Left array still contains elements
        arr[k++] = arrLeft[i++];                // put them in the real array
    }

    while (j < sizeRight) {                     // If the right array still contains elements
        arr[k++] = arrRight[j++];               // put them in the real array
    }
}

void mergeSort(int *arr, int size){
    int *arrLeft;                               // Initializing two temporary dynamic
    int *arrRight;                              // arrays to hold the two arrays to be merged

    int sizeLeft;                               // The two arrays' sizes
    int sizeRight;

    if (size > 1) {                             // checking if the size of the arrays divide bigger than 1

        if (size % 2 == 0){                     // if the size is even divide both arrays into 2 equal parts
            sizeLeft = size / 2;
            sizeRight = size / 2;
        }
        else{                                   // else if the size is odd divide them so that the left array would have 1 more number than the right one
            sizeLeft = (size / 2) + 1;
            sizeRight = size / 2;

```

```
      }

        arrLeft = new int[sizeLeft];            // Give the Left array its size
        arrRight = new int[sizeRight];          // Give the Right array its size

        /************/
        /* Dividing */
        /************/

        for (int i = 0; i < sizeLeft; i++){                        // Putting the first half of the numbers of the real array in the temporary left array
            arrLeft[i] = arr[i];
        }

        int j = 0;                                                  // j works as an index number for the temporary Right array

        for (int i = sizeLeft; i < size; i++){                     // Putting the second half of the numbers of the real array in the temporary right array
            arrRight[j++] = arr[i];
        }

        /*************/
        /* Recursion */
        /*************/

        mergeSort(arrLeft, sizeLeft);                               // Recursively call the Merge Sort function for the Left array untill the size is less than or equals 1
        mergeSort(arrRight, sizeRight);                             // Recursively call the Merge Sort function for the Right array untill the size is less than or equals 1
        merge(arr, arrLeft, arrRight, sizeLeft, sizeRight, size);  // Call the Merge function to sort and merge both halfs of the array

    }
}

int MergeSort(){

    mergeSort(Arr, size);                                           // Calling the mergeSort Function

    mergeTotal = (clock() - mergeStart) / (double)CLOCKS_PER_SEC; // Calculating the mergeSort Run-time in seconds

    cout << endl;
    cout << "Merge Sort Completed!" << endl;
    return 0;
}

/**************/
/* Quick Sort */
/**************/

void Swap(int &arr1, int &arr2){

    int temp;                       /*********************/
    temp = arr1;                    /*A function to swap */
    arr1 = arr2;                    /*  any two integers */
    arr2 = temp;                    /*    by refrence    */
}                                   /*********************/

```

```
int partition(int *arr, int firstIndex, int lastIndex){     // The partition function takes the array that should be sorted, the first and last indexes of the elements of the array
    int i = firstIndex - 1;                                 // An index that starts before the first element of the part that needs to be sorted
    int pivot = arr[lastIndex];                             // The last element in the partition in which we compare the other element with
    for (int j = firstIndex; j < lastIndex; j++){           // A loop that goes through all the elements in the partition
        if (arr[j] <= pivot){                               // If the element of the array in index j is less than or equal the specified pivot
            i++;                                            // i gets incremented
            Swap(arr[i], arr[j]);                           // then the elemets of the array in index i and j gets swapped
            counterQuick++;
        }
    }
    Swap(arr[i + 1], arr[lastIndex]);                       // Swaps the pivot with the element at i+1 to put the pivot at its rightful place
    return i + 1;                                           // Returns the new pivot's position to go through the following partitions
}
void quickSort(int *arr, int firstIndex, int lastIndex){    // The quick sort function takes the array that should be sorted, the first and last indexes of the elemnts that needs to be sorted
    if (firstIndex < lastIndex){                            // Checks if in the two sent indexes, the first is less than the last
        int p = partition(arr, firstIndex, lastIndex);      // Calls partition function to sort this partition
        quickSort(arr, firstIndex, p - 1);                  // Recursively calls itself in order to sort the first part of the array
        quickSort(arr, p + 1, lastIndex);                   // Recursively calls itself in order to sort the second part of the array
    }
}
int QuickSort(){
    quickSort(Arr2, 0, size - 1);                                   // Calling the quickSort Function
    quickTotal = (clock() - quickStart) / (double)CLOCKS_PER_SEC;   // Calculating the quickSort Run-time
    cout << endl;
    cout << "Quick Sort Completed!" << endl;
    return 0;
}
int main(){
```

```
    //ios_base::sync_with_stdio(false); // to have the cout and cin run faster

    cout << "Enter size of Array: ";
    cin >> size;                        // Aquiring the size of the array from the user

    cout << endl;

    //srand(time(NULL)); //makes sure that numbers are not the same in each run

    Arr = new int[size];                // Giving the dynamic array its size
    Arr2 = new int[size];

    for (int i = 0; i < size; i++){     // A loop to let the user enter Elements to be sorted

        cout << "Enter number: ";
        cin >> Arr[i];
        //Arr[i] = rand() % 100; // generates random numbers;
    }

    cout << endl;

    cout << "unsorted array :";
    for (int i = 0; i < size; i++){

        Arr2[i] = Arr[i];
        cout << Arr2[i]<<" ";
    }



    mergeStart = clock();
    thread Merge(MergeSort);            // A function for displaying the sorted numbers in the array by the mergeSort algorithm
    quickStart = clock();
    thread Quick(QuickSort);            // A function for displaying the sorted numbers in the array by the quickSort algrithm




    Merge.join();
    Quick.join();



    display(Arr, size, 1);
    display(Arr2, size, 2);
    cout << "Time Elapsed for Merge: " << fixed << setprecision(8) << mergeTotal << "s" << endl;
    cout << "Time Elapsed for Quick: " << fixed << setprecision(8) << quickTotal << "s" << endl;

    //printf("Time Elapsed for Merge: %.22lf ms \n", mergeTotal*1000); // Printing out the run-time of the program
    //printf("Time Elapsed for Quick: %.22lf ms \n", quickTotal*1000); // Printing out the run-time of the program

    return 0;
}
```

2. Complexity:

Merge Sort:

Partition step is of number of steps: c

Recursive calls is of number of steps: 2*F(n/2)

Merging is of number of steps: n

So the recurrence formula will be as follows:

F(n) = 2*F(n/2) + cn

According to the master theorem

F(n) = aF(n/b) + nd then

Then a = 2, b =2, and d=1

Then (a = 2) = (bd=21=2) which is the second case : F(n) = O(nd log2 n)

The F(n) = O("nd log2 n") = O("n1 log2 n") = O("n∗log2 n")

Worst- case performance: O(*n* log *n*)

Best-case performance: O(*n* log *n*) typical, O(*n*) natural variant

Average performance: O(*n* log *n*)

Quick Sort:

Best case has partitions are as evenly balanced as possible: their sizes either are equal of size ((n-1)/2) if the number of elements is odd or are within 1 of each other if the number of elements is even. So the recursive call on both partitions is of complexity 2*c*(n/2)=cn, then 22*c(n/22)=cn, then 23*c(n/23)=cn, and so on.

The algorithm formula is f(n)=2*f(n/2)+cn

Worst case has the most unbalanced partitions possible. The sequence of n elements S(n) in this case is partitioned to {pivot, S(n-1)}. So the recursive call on S(n) is of complexity cn, and the recursive call on S(n-1) is of complexity c(n-1), and the recursive call on S(n-2) is of complexity c(n-2), and so on.

The algorithm formula is f(n)=f(n-1) + cn

Worst- case performance: O(n^2)

Best-case performance: O(*n* log *n*)

Average performance: O(*n* log *n*)

3. Comparison between both algorithms:

Algorithms	Number of Steps	Which generates first	Test Case
Merge Sort	According to our test case #1 the number of steps of the Merge Sort: 538 steps The Merge sort is slower than the Quick sort in smaller test-cases Worst-case: O(n*log(n))	According to our test case the duration of the run time of the merge sort function is 0.003s but the duration of the quick sort function takes less than 1/100000000s that the time is showing as zero seconds. In agreement with tried out test cases; the Merge sort works best with very large test cases than the quick sort does	Test Case #1: Enter size of Array: 100 unsorted array :88 74 27 97 98 57 81 88 38 4 79 83 72 33 77 59 66 6 96 37 95 27 36 42 75 32 58 99 28 10 81 96 24 65 87 83 21 4 77 71 74 48 67 25 16 18 64 1 79 64 6 30 87 66 36 8 65 93 57 6 30 15 65 31 77 0 57 59 27 49 99 6 12 35 74 69 65 30 91 98 70 39 49 29 49 19 43 2 62 32 91 49 55 33 33 0 31 21 12 32 Quick Sort Completed! Merge Sort Completed! MergeSorted: 0 0 1 2 4 4 6 6 6 6 8 10 12 12 15 16 18 19 21 21 24 25 27 27 27 28 29 30 30 30 31 31 32 32 32 33 33 33 35 36 36 37 38 39 42 43 48 49 49 49 49 55 57 57 57 58 59 59 62 64 64 65 65 65 65 66 66 67 69 70 71 72 74 74 74 75 77 77 77 79 79 81 81 83 83 87 87 88 88 91 91 93 95 96 96 97 98 98 99 99 Merge Sort Counter: 538 QuickSorted: 0 0 1 2 4 4 6 6 6 6 8 10 12 12 15 16 18 19 21 21 24 25 27 27 27 28 29 30 30 30 31 31 32 32 32 33 33 33 35 36 36 37 38 39 42 43 48 49 49 49 49 55 57 57 57 58 59 59 62 64 64 65 65 65 65 66 66 67 69 70 71 72 74 74 74 75 77 77 77 79 79 81 81 83 83 87 87 88 88 91 91 93 95 96 96 97 98 98 99 99 Quick Sort Counter: 361 Time Elapsed for Merge: 0.00200000s Time Elapsed for Quick: 0.00100000s
	Consequently the number of steps of the quick sort function: 361 steps	According to our test cases the quick sort	Test Case #2: Enter size of Array: 50 unsorted array: 3 14 6 42 94 23 60 58

Quick Sort	The quicksort is very fast with smaller cases Worst case: $O(n^2)$ Which is so much worse than the merge sort's worst case	Quick sort is typically faster than merge sort. However, when the data set is huge or if the array is already sorted, merge sort is the clear winner in terms of speed.	83 80 64 85 56 47 42 80 71 17 47 6 53 84 59 19 60 16 86 48 6 36 17 28 62 67 90 23 19 22 41 57 31 39 37 85 6 18 26 26 2 75 Quick Sort Completed! Merge Sort Completed! MergeSorted: 2 3 6 6 6 6 14 16 17 17 18 19 19 22 23 23 26 26 28 31 36 37 39 41 42 42 47 47 48 53 56 57 58 59 60 60 62 64 67 71 75 80 80 83 84 85 85 86 90 94 Merge Sort Counter: 229 QuickSorted: 2 3 6 6 6 6 14 16 17 17 18 19 19 22 23 23 26 26 28 31 36 37 39 41 42 42 47 47 48 53 56 57 58 59 60 60 62 64 67 71 75 80 80 83 84 85 85 86 90 94 Quick Sort Counter: 167 Time Elapsed for Merge: 0.00200000s Time Elapsed for Quick: 0.00100000s

4. Why is one algorithm is better than the other?

The quick sort works best when cases are small or needs lots of sorting, as it takes much less time and steps than the merge sort. However, the merge sort works best with very large cases as it will take less time and steps than the quick sort.