#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;

int counterQuick = 0;

clock\_t quickStart;

clock\_t mergeStart;

double quickTotal;

double mergeTotal;

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 flag){

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, 1); // Recursively call the Merge Sort function for the Left array untill the size is less than or equals 1

mergeSort(arrRight, sizeRight, 1); // 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

}

if (!flag)

mergeTotal = (clock() - mergeStart) / (double)CLOCKS\_PER\_SEC;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* 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, bool flag){ // 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, 1); // Recursively calls itself in order to sort the first part of the array

quickSort(arr, p + 1, lastIndex, 1); // Recursively calls itself in order to sort the second part of the array

}

if (!flag)

quickTotal = (clock() - quickStart) / (double)CLOCKS\_PER\_SEC;

}

int main() {

ios\_base::sync\_with\_stdio(false); // Fast input output

cout << "Enter size of Array: ";

cin >> size; // Aquiring the size of the array from the user

cout << endl;

srand(time(NULL));

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

Arr[i] = rand() % 100;

}

cout << endl;

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

Arr2[i] = Arr[i];

}

mergeStart = clock();

thread Merge(mergeSort, Arr, size, 0);// A function for displaying the sorted numbers in the array by the mergeSort algorithm

quickStart = clock();

thread Quick(quickSort, Arr2, 0, size - 1, 0);// 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 << fixed << setprecision(8) << mergeTotal << " " << setprecision(8) << quickTotal << endl;

// printf("Time Elapsed for Merge: %.2fs \n", mergeTotal); // Printing out the execution time of the program

// printf("Time Elapsed for Quick: %.2fs \n", quickTotal); // Printing out the execution time of the program

return 0;

}