#include <iostream>

#include <vector>

#include <cstdlib>

#include <chrono>

using namespace std;

// Function to print the elements of an array

void print(const vector<int>& arr) {

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

        cout << arr[i] << " ";

    }

    cout << endl;

}

// Merge function used in MergeSort to merge two halves

void merge(vector<int>& arr, int left, int mid, int right) {

    int n1 = mid - left + 1;  // Size of the left subarray

    int n2 = right - mid;     // Size of the right subarray

    vector<int> L(n1), R(n2); // Temporary arrays to hold the two halves

    // Copy data to temporary arrays L[] and R[]

    for (int i = 0; i < n1; i++)

        L[i] = arr[left + i];

    for (int j = 0; j < n2; j++)

        R[j] = arr[mid + 1 + j];

    int i = 0, j = 0, k = left; // Initial indexes of subarrays and merged array

    // Merge the temporary arrays back into arr[left..right]

    while (i < n1 && j < n2) {

        if (L[i] <= R[j]) {

            arr[k] = L[i];

            i++;

        } else {

            arr[k] = R[j];

            j++;

        }

        k++;

    }

    // Copy the remaining elements of L[], if any

    while (i < n1) {

        arr[k] = L[i];

        i++;

        k++;

    }

    // Copy the remaining elements of R[], if any

    while (j < n2) {

        arr[k] = R[j];

        j++;

        k++;

    }

}

// MergeSort function to sort an array using the merge sort algorithm

void mergeSort(vector<int>& arr, int left, int right) {

    if (left < right) {

        int mid = left + (right - left) / 2;

        // Sort the first and second halves

        mergeSort(arr, left, mid);

        mergeSort(arr, mid + 1, right);

        // Merge the sorted halves

        merge(arr, left, mid, right);

    }

}

// Partition function used in QuickSort to partition the array around a pivot

int partition(vector<int>& arr, int low, int high) {

    int pivot = arr[high]; // Pivot element

    int i = low - 1;       // Index of smaller element

    // Rearrange elements based on pivot

    for (int j = low; j < high; j++) {

        if (arr[j] < pivot) {

            i++;

            swap(arr[i], arr[j]);

        }

    }

    swap(arr[i + 1], arr[high]);

    return i + 1;

}

// QuickSort function to sort an array using the quick sort algorithm

void quickSort(vector<int>& arr, int low, int high) {

    if (low < high) {

        int pi = partition(arr, low, high);

        // Recursively sort elements before and after partition

        quickSort(arr, low, pi - 1);

        quickSort(arr, pi + 1, high);

    }

}

int main() {

    ios\_base::sync\_with\_stdio(false);

    // Initialize an array with 300 random integers between 1 and 1000

    vector<int> arr(300);

    for (int i = 0; i < 300; i++)

        arr[i] = 1 + rand() % 1000;

    // Create a copy of the array for quicksort

    vector<int> arr\_copy = arr;

    // Print the given array

    cout << "Given array is" << endl;

    print(arr);

    // Measure and print the time taken by MergeSort

    auto start\_time = chrono::high\_resolution\_clock::now();

    mergeSort(arr, 0, arr.size() - 1);

    auto stop\_time = chrono::high\_resolution\_clock::now();

    chrono::duration<double> elapsed = stop\_time - start\_time;

    cout << "\n\nSorted array using MergeSort is -" << endl;

    print(arr);

    cout << fixed;

    cout << "\nElapsed time for MergeSort: " << elapsed.count() << " s\n";

    // Measure and print the time taken by QuickSort

    start\_time = chrono::high\_resolution\_clock::now();

    quickSort(arr\_copy, 0, arr\_copy.size() - 1);

    stop\_time = chrono::high\_resolution\_clock::now();

    elapsed = stop\_time - start\_time;

    cout << "\nSorted array using QuickSort is -" << endl;

    print(arr\_copy);

    cout << "\nElapsed time for QuickSort: " << elapsed.count() << " s\n";

    return 0;

}