```
// Name: Connor Marler
// Instructor: Gerry Howser
// Date: 10/13/2022
// Project 2 Merge Sort Analysis

//Source: https://slaystudy.com/c-program-to-implement-merge-sort-using-templates/

#include<iostream>
//#include<vector>
#include <chrono>
#include <cassert>

using namespace std;


//Purpose: template function to merge 2 componenets of the array, arr
//PreCondition: takes in an templated array and integer start and end points
reflecting the specific subarray
//PostCondition: the result is written back to the original array
//Invariant:(indexOfMergedArray == left) && (indexOfSubArrayOne == 0) &&
(indexOfSubArrayTwo == 0))
template<typename T>
void Merge(T array[], int const left, int const right)
{
    auto mid = (left + right) / 2;
    auto const subArrayOne = mid - left + 1;
    auto const subArrayTwo = right - mid;

    // Create temp arrays
    auto *leftArray = new int[subArrayOne],
         *rightArray = new int[subArrayTwo];

    // Copy data to temp arrays leftArray[] and rightArray[]
    for (auto i = 0; i < subArrayOne; i++)
        leftArray[i] = array[left + i];
    for (auto j = 0; j < subArrayTwo; j++)
        rightArray[j] = array[mid + 1 + j];

    auto indexOfSubArrayOne
        = 0, // Initial index of first sub-array
        indexOfSubArrayTwo
        = 0; // Initial index of second sub-array
    int indexOfMergedArray
        = left; // Initial index of merged array

    // Merge the temp arrays back into array[left..right]
    assert((indexOfMergedArray == left) && (indexOfSubArrayOne == 0) &&
(indexOfSubArrayTwo == 0));
    while (indexOfSubArrayOne < subArrayOne && indexOfSubArrayTwo < subArrayTwo)
    {
      if (leftArray[indexOfSubArrayOne] <= rightArray[indexOfSubArrayTwo])
      {
        array[indexOfMergedArray] = leftArray[indexOfSubArrayOne];
        indexOfSubArrayOne++;
      }
      else {
        array[indexOfMergedArray] = rightArray[indexOfSubArrayTwo];
        indexOfSubArrayTwo++;
      }
      indexOfMergedArray++;
    }
    // Copy the remaining elements of
    // left[], if there are any
    while (indexOfSubArrayOne < subArrayOne)
    {
      array[indexOfMergedArray] = leftArray[indexOfSubArrayOne];
      indexOfSubArrayOne++;
      indexOfMergedArray++;
    }
    // Copy the remaining elements of
    // right[], if there are any
    while (indexOfSubArrayTwo < subArrayTwo)
    {
      array[indexOfMergedArray] = rightArray[indexOfSubArrayTwo];
      indexOfSubArrayTwo++;
      indexOfMergedArray++;
    }
    delete[] leftArray;
    delete[] rightArray;
}

//Purpose: template function to perform merge sort on array, arr
//PreCondition: takes in a templated array and integer start and end points
reflecting the length of the array
//PostCondition: the list is now sorted in ascending orders
template<typename T>
void MergeSort(T arr[], int start, int end)
{
  if (start < end)
  {
    int mid = (start + end) / 2;
    MergeSort(arr, start, mid); // merge sort the elements in range [start, mid]
    MergeSort(arr, mid + 1, end); // merge sort the elements in range [mid+1, end]
    Merge(arr, start, end);   // merge the above 2 componenets
  }
}


//Purpose: Template function to print array
//PreCondition: takes in an array and a value of the size of the array
//PostCondition: outputs the array givens
template<typename T>
void PrintArray(T arr[], int n)
{
    for (int i = 0; i < n; ++i)
        cout << arr[i] << " ";
    cout << "\n\n";
}


//Purpose: take in an array and fill it with a predetermined element set
//PreCondition: must receive arr[size] and a string value determining how to fill
the set
//PostCondition: must fill arr[size] with the values in the appropriate as
determined by preSortOrder
void populateArray(int *arr, const int size, const string preSortOrder)
{
  // This if else tree could also be re-written as a switch case
  if(preSortOrder == "assPreSortOrder")
  {
    for(int i = 0; i <= size; i++)
      {
        arr[i] = i + 1;
      }
  }
  else if(preSortOrder == "decPreSortOrder")
  {
    for(int j = 0; j <= size; j++)
      {
        arr[j] = size - j;
      }
  }
  else if(preSortOrder == "randPreSortOrder")
  {
    for(int k = 0; k <= size; k++)
      {
        arr[k] = rand() % size + 1;
      }
  }
  else // error handling
  {
    cout << "ERROR: Something broke. Please quit and try again." << endl;
  }
}

int main()
{

  // These are what we will use for the time measurement
  using chrono::high_resolution_clock;
  using chrono::duration_cast;
  using chrono::duration;
  using chrono::nanoseconds;

  srand (time(NULL));
  const int SIZE = 500;
  string preSortOrder;
  int innerLoop = 1000;
  int outerLoop = 10;
  int averageTime = 0;
  int time = 0;


  int mainArray[SIZE]; // This the hard, original copy of the array
  int workingArray[SIZE]; // This is the copy used to work with and innerLoop
through


  cout << "MERGE SORT AVG TIMES" << endl;
  for(int i = 0; i <= 2; i ++) // loops through each case
  {
    if(i == 0)
    {
      preSortOrder = "assPreSortOrder";
      cout << endl << "Ascending Order Time:" << endl;
    }
    else if(i == 1)
    {
      preSortOrder = "decPreSortOrder";
      cout << endl << "Descending Order Time:" << endl;
    }
    else if(i == 2)
    {
      preSortOrder = "randPreSortOrder";
      cout << endl << "Random Order Time:" << endl;
    }

    populateArray(mainArray, SIZE, preSortOrder);
    averageTime = 0;

    for(int k = 0; k < outerLoop; k++) // takes 10 data point
    {
      averageTime = 0;
      for(int j = 0; j < innerLoop; j++) // perform the sort 1000 times and take an
average for one data point
      {
        for (int a = 0; a < SIZE; a++) // This copies over the original array onto
a working copy
        {
          workingArray[a] = mainArray[a];
        }
        //cout << "Before Sort" << endl;
        //PrintArray(workingArray, SIZE);
        auto time1 = high_resolution_clock::now(); // take initial time
        MergeSort(workingArray, 0, SIZE - 1);         // do the sort
        auto time2 = high_resolution_clock::now(); // time the after time
        //cout << "After Sort" << endl;
        //PrintArray(workingArray, SIZE);
        auto nanoSeconds = duration_cast<nanoseconds>(time2 - time1); // find the
difference of the times
        time = nanoSeconds.count();  // convert to an int
        averageTime += time;
      }
      averageTime /= innerLoop;
      cout << "Trial # " << k + 1 << ": " << averageTime << " ns" << endl;
    }
  }
}
```