```
/*****************************************************************************
 *
 * James A. Avery
 * 50189091
 * CSCI 532 Semester Project
 *
 * Problem #3 (expanded)
 * Write a C/C++/Matlab/Java program to compute insertion sort and merge sort.
 * (You can use the code from the textbooks by Deitel and Deitel or from the
 *  web.)
 * Obtain the run time of both routines.
 * The input data should be an int array containing random element values
 *  (between, say, 0 and 1023).
 * Obtain run time T with 1D (input) array size of n=16, 256, 4096, 65536,
 *  1048576 (which equal 2^p, where p = 4, 8, 12, 16, 20).
 * The run time for each n should be average with about floor(512/(p*p)) runs.
 * Each run (for a given n) should use a different random input.
 * Plot (with Excel, Matlab, or other available tools) the run time for both
 *  routines on one plot, with the x axis in p values, and y axis in log(T).
 * Label on the plot which curve is for insertion sort and which is for merge
 *  sort.
 * Submit C/C++ programs and plot, with instructions in readme.txt on how to
 *  build and run the program. (Include the Dev-C++ or MS Studio or Java
 *  NetBeans project file.)
 *
 *****************************************************************************/

/**  References  ************************************************************
 *
 *  Time calculation in C++11
 *      https://solarianprogrammer.com/2012/10/14/cpp-11-timing-code-performance/
 *
 *  Random numbers in C++11
 *      http://en.cppreference.com/w/cpp/numeric/random
 *
 *  Using std::copy instead of a for loop for a single-line vector print:
 *      https://stackoverflow.com/questions/10750057/how-to-print-out-the-contents-of-a-
vector/11335634#11335634
 *
 *  Merge Sort from Deitel and Deitel
 *  Insertion Sort from Deitel and Deitel
 *
*****************************************************************************/

#include <iostream>
#include <fstream>
#include <vector>
#include <random>
#include <iterator>
#include <chrono>
#include <string>
#include <functional>
#include <algorithm>    // For remove_if when setting the save file name


/*
By default, the program runs a decreasing number of trials, such that for an
    array of size n=2^p, the number of trials is floor(512/(p*p)), (per the
    project instructions).
I would like a more scientific approach (i.e. a more) uniform number of
```

```
    trials), so I have included a command line option to run the same number of
    trials for each vector size. On my computer, a vector of n = 2^20 = 1048567
    can be sorted in around 24 minutes using insertion sort, so a batch size of
    24 allows the entire program to run roughly overnight.
*/
constexpr int STANDARD_BATCH_SIZE = 24;

/*
If, instead, the program is being run as assigned, then calculate the batch
    size based on the formula floor(512/p^2)
*/
inline int assigned_batch_size( int p ) {
    return floor ( 512 / ( p * p ) );
}

/*
Arrays of size n = 20 take almost half an hour to run, which is far too long
    for testing. This limits the size of arrays to those that can be processed
    quickly.
*/
constexpr int MAX_N = 1000000;

// Struct to allow passing both a function and some name for it
struct NamedFunction {
    std::string function_name;
    std::function<void( std::vector<int>& )> function;
};



// Sort a vector in place using insertion sort
void insertion_sort( std::vector<int>& );


// Sort a vector in place using merge sort
void merge_sort( std::vector<int>& );
// Merge sort helper functions
void merge( std::vector<int>&, int, int, int, int );
void sort_sub_vector( std::vector<int>&, int, int );


// Fill an int vector with random integers
void random_fill( std::vector<int>& );


// Overload operator<< for vectors to simplify output of all vector elements
template<typename T>
std::ostream & operator<<( std::ostream&, const std::vector<T>& );


// Function that allows each different sorting algorithm to be tested and timed
//  using the same code.
void test_sorter( NamedFunction&, std::vector<int>, bool, bool );

// Demonstrate that a given sorter function actually does sort a vector properly
void demo_sorter( NamedFunction );
```

```
120 int main(int argc, char **argv) {
121     // Put the command line arguments into a vector for easier access
122     std::vector<std::string> args{argv + 1, argv + argc};
123 
124     // Flags to determine run-time behavior.
125     // Default values are the problem as assigned.
126     // Changing these through command line options results in more interesting tests.
127     bool run_insertion_sort = true;
128     bool run_merge_sort = true;
129     bool limit_n = false;
130     bool demo = false;
131     bool time = true;
132 
133     // p determines both the array size and the number of tests per array
134     // Arrays of size 2^p will be tested either floor(512/(p*p)) times or 32 times
135     // Having a vector of ps will make things easier later
136     std::vector<int> ps { 4, 8, 12, 16, 20  };
137 
138     // Determine the sizes of the test runs.
139     // If false, then each vector of size 2^p will be tested floor(512/(p*p)) times
140     // If true, each vector will be tested the same number of times (set above)
141     bool equal_batch_sizes = false;
142 
143 
144     // Adjust run-time flags according to command line arguments
145     for ( auto flag : args ) {
146         if ( flag == "merge-only" ) { run_insertion_sort = false; }
147         if ( flag == "insertion-only" ) { run_merge_sort = false; }
148         if ( flag == "equal-batches" ) { equal_batch_sizes = true; }
149         if ( flag == "limit-n" ) { limit_n = true; }
150         if ( flag == "demo-sorters" ) { demo = true; }
151         if ( flag == "no-time" ) { time = false; }
152     }
153 
154     // Create a vector of functions to test based on command line options
155     //  Default is both Insertion Sort and Merge Sort,
156     //  but these can be stopped by command line options above
157     std::vector<NamedFunction> sorters;
158     if ( run_insertion_sort ) {
159         sorters.push_back( NamedFunction{ "Insertion Sort",  insertion_sort } );
160     }
161     if ( run_merge_sort ) {
162         sorters.push_back( NamedFunction{ "Merge Sort", merge_sort } );
163     }
164 
165     // Demonstrate that the sort functions sort correctly (disabled by default)
166     if ( demo ) {
167         for ( auto sorter : sorters ) {
168             demo_sorter ( sorter );
169         }
170     }
171 
172     // The heart of the project
173     // Test the run-time of each sort function as assigned (enabled by default)
174     if ( time ) {
175         for ( auto sorter : sorters ) {
176             test_sorter( sorter, ps, equal_batch_sizes, limit_n );
177         }
178     }
179 
```

```
    return 0;
}



// Sort a vector in place using insertion sort
void insertion_sort( std::vector<int> &v ) {
    // Loop throught the elements of the array
    for ( int j = 1; j < v.size(); j++ )
    {
        int key = v[j];
        int i = j - 1;

        while ( i >= 0 && v[i] > key )
        {
            v[ i + 1 ] = v[i];
            i -= 1;
        }

        v[ i + 1 ] = key;
    }
}



// Sort a vector in place using merge sort
void merge_sort( std::vector<int> &v ) {
    sort_sub_vector( v, 0, v.size() - 1 );
}



// Helper function to merge subvectors
void merge( std::vector<int> &v, int left, int mid1, int mid2, int right ) {
    int left_index = left;
    int right_index = mid2;
    int combined_index = left;
    std::vector<int> combined( v.size() );

    // Merge subvectors until reaching the end of either
    while ( left_index <= mid1 && right_index <= right ) {
        // Place the smaller of the two current elements into result
        //  and move to next space invector
        if ( v[ left_index ] <= v[ right_index ] ) {
            combined[ combined_index++ ] = v[ left_index++ ];
        } else {
            combined[ combined_index++ ] = v[ right_index++ ];
        }
    }

    // Put any leftover elements into the combined vector
    if ( left_index == mid2 ) { // if at the end of the left vector
        while ( right_index <= right ) {    // copy rest of right vector
            combined[ combined_index++ ] = v[ right_index++ ];
        }
    } else {                    // if at the end of the right vector
        while ( left_index <= mid1 ) {      // copy rest of left vector
            combined[ combined_index++ ] = v[ left_index++ ];
        }
    }
```

```
240
241     // Copy values back into the original vector
242     for ( int i = left; i <= right; i++ ) {
243         v[i] = combined[i];
244     }
245 }
246
247
248
249 // Helper function to recursively sort sub-vectors
250 void sort_sub_vector( std::vector<int> &v, int low, int high ) {
251     // Test against base case where size of vector is 1
252     if ( ( high - low ) >= 1 ) {    // if NOT base case then
253
254         // Calculate midpoint of the vector,
255         //  and the next element to the right.
256         int mid1 = ( low + high ) / 2;
257         int mid2 = mid1 + 1;
258
259         // Split vector in half and sort each half recursively
260         sort_sub_vector( v, low, mid1 );   // left half
261         sort_sub_vector( v, mid2, high );  // right half
262
263         // Merge the two sorted vectors
264         merge( v, low, mid1, mid2, high );
265     } // end if not base case
266 }
267
268
269
270 // Fill a vector in place with random integers
271 void random_fill( std::vector<int> &v ) {
272     // Initialize the C++11 random device
273     static std::random_device rd{};
274     static std::mt19937 mt{ rd() };
275     static std::uniform_int_distribution<int> dist{ 1, 1024 };
276
277     for ( int i = 0; i < v.size(); i++ )
278     {
279         v[i] = dist( mt );
280     }
281 }
282
283
284
285 // Overload operator<< for vectors to simplify output of all vector elements
286 template <typename T>
287 std::ostream& operator<< ( std::ostream &out, const std::vector<T> &v ) {
288     std::copy( v.begin(), v.end(), std::ostream_iterator<T>( out, " " ) );
289     return out;
290 }
291
292
293
294 // Given a NamedFunction, a vector of values for p, and booleans to determine
295 //  whether or not to process equal batch sizes and whether or not to limit
296 //  the size of the vector to vectors that can be sorted quickly, calculate the
297 //  execution times for each of the sort algorithms
298 void test_sorter( NamedFunction &f, std::vector<int> ps,
299                     bool equal_batch_sizes,
```

```
                    bool limit_n ) {
    // Create a file to output the results
    std::string file_name{ f.function_name };
    file_name.erase( std::remove_if( file_name.begin(), file_name.end(), isspace ) );
    std::transform( file_name.begin(), file_name.end(), file_name.begin(), tolower );
    file_name += ".csv";

    std::ofstream fout( file_name );

    std::cout << std::endl << f.function_name << std::endl;

    // Iterate through each of the p values (i.e. p = 4, 8, 12, ...)
    for ( int p : ps ) {
        // Vector size
        int n = pow( 2, p );

        // If n is being limited to certain fast-running small values,
        //  then skip this iteration
        if ( limit_n && n > MAX_N ) { continue; }

        // Calculate batch size
        int batch_size = equal_batch_sizes ?
                         STANDARD_BATCH_SIZE :
                         assigned_batch_size( p );

        // First column of the output is the size of the vector
        std::cout << n << "\t" << std::flush;
        fout << n << "\t" << std::flush;

        for ( int trial = 0; trial < batch_size; trial++ ) {
            // Create a vector and fill it with random numbers
            std::vector<int> v( n );
            random_fill( v );

            // Time execution of the sort
            auto start = std::chrono::steady_clock::now();
            f.function( v );
            auto end = std::chrono::steady_clock::now();

            std::cout
                << std::chrono::duration<unsigned long long, std::nano>(end - start).count()
                << "\t" << std::flush;
            fout
                << std::chrono::duration<unsigned long long, std::nano>(end - start).count()
                << "\t" << std::flush;
        }

        std::cout << std::endl;
        fout << std::endl;
    }

    fout.close();
}



// Demonstrate that the sorter actually does sort a vector properly
void demo_sorter( NamedFunction sorter ) {
    std::cout << std::endl << "=====  Demonstrating "
            << sorter.function_name
```

```
360             << "  =====" << std::endl;
361 
362     for ( int i = 0; i < 10; i++ ) {
363         std::vector<int> v( 10 );
364         random_fill( v );
365         std::cout << std::endl << "Before: " << v << std::endl;
366         sorter.function(v);
367         std::cout << "After: " << v << std::endl;
368     }
369 }
```