CSE 330 LABORATORY -- Week 10, Spring 2018

Instructor: Kerstin Voigt

In this lab you will implement and test three well known sorting algorithms: insertionSort, selectionSort, and mergeSort. The insertionSort and selectionSort algorithms are of computational complexity $O(N^2)$. Convince yourself of this fact when implementing the steps of the algorithm. These two algorithms are considered inefficient. The mergeSort algorithm is known as an efficient sorting algorithm with complexity O(N log N). Again, understand what accounts for the more efficient performance of mergeSort when you go through the steps of implementation.

Exercise 1: In a file Sort.h, implement all three sorting algorithms as template functions. You are allowed to use the STL data structure vector with '#include <vector>'. Proceed to "mindfully" copy the code below. Use your own judgment whether to copy or omit any of the lines of comment (/*…*/ or // …).

```
// Sort.h
// adopted from Weiss, Data Structures and Alg Analysis with C++

#ifndef SORT_H
#define SORT_H

#include <vector>
using namespace std;

template <typename C>
void print_vec(vector<C> v)
{
  for (int i = 0; i < v.size(); i++)
    cout << v[i] << " ";
  cout << endl;
}

// Simple insertion sort
template <typename Comparable>
void insertionSort( vector<Comparable> & a )
{
    for( int p = 1; p < a.size( ); ++p )
    {
        Comparable tmp = std::move( a[ p ] );

        int j;
        for( j = p; j > 0 && tmp < a[ j - 1 ]; --j )
            a[ j ] = std::move( a[ j - 1 ] );
        a[ j ] = std::move( tmp );
    }
}
```

```
template <typename Comparable>
void selectionSort(vector<Comparable> & a)
{
  for (int i = a.size() - 1; i > 0; i--)
    {
      int largepos = 0;
      for (int j = 1; j <= i; j++)
        {
            if (a[largepos] < a[j])
                  largepos = j;
        }
      if (i != largepos)
            std::swap(a[i],a[largepos]);
  }
}

// a prototypes for functions below ...
template <typename Comparable>
void merge(vector<Comparable>&, vector<Comparable>&, int, int, int);

template <typename Comparable>
void mergeSort(vector<Comparable> &, vector<Comparable> &, int, int);

// Mergesort algorithm (driver, top-level);

template <typename Comparable>
void mergeSort( vector<Comparable> & a )
{
  vector<Comparable> tmpVec( a.size( ) );

  mergeSort( a, tmpVec, 0, a.size( ) - 1 );
}


/* Internal method that makes recursive calls.
 * a is an array of Comparable items.
 * tmpVec is an array to place the merged result.
 * left is the left-most index of the subarray.
 * right is the right-most index of the subarray.
 */
template <typename Comparable>
void mergeSort( vector<Comparable> & a,
                vector<Comparable> & tmpVec, int left, int right )
{
    if( left < right )
    {
      int center = ( left + right ) / 2;
        mergeSort( a, tmpVec, left, center );
        mergeSort( a, tmpVec, center + 1, right );
        merge( a, tmpVec, left, center + 1, right );
    }
}
```

```
/* Internal method that merges two sorted halves of a subarray.
 * a is an array of Comparable items.
 * tmpVec is an array to place the merged result.
 * leftPos is the left-most index of the subarray.
 * rightPos is the index of the start of the second half.
 * rightEnd is the right-most index of the subarray.
 */
template <typename Comparable>
void merge( vector<Comparable> & a, vector<Comparable> & tmpVec,
            int leftPos, int rightPos, int rightEnd )
{
    int leftEnd = rightPos - 1;
    int tmpPos = leftPos;
    int numElements = rightEnd - leftPos + 1;

    while( leftPos <= leftEnd && rightPos <= rightEnd )
      {
        if( a[ leftPos ] <= a[ rightPos ] )
            tmpVec[ tmpPos++ ] = std::move( a[ leftPos++ ] );
        else
            tmpVec[ tmpPos++ ] = std::move( a[ rightPos++ ] );
      }

    while( leftPos <= leftEnd )    // Copy rest of first half
        tmpVec[ tmpPos++ ] = std::move( a[ leftPos++ ] );

    while( rightPos <= rightEnd )  // Copy rest of right half
        tmpVec[ tmpPos++ ] = std::move( a[ rightPos++ ] );

    // Copy tmpVec back
    for( int i = 0; i < numElements; ++i, --rightEnd )
        a[ rightEnd ] = std::move( tmpVec[ rightEnd ] );
}

#endif
```

Exercise 2: Write your own test program in file SortMain.cpp which generates an unordered vector of 20 random integers and two copies of this vector. The program is to sort the first vector copy with insertionSort, the second vector copy with selection Sort, and the third copy with mergeSort. The test program should print the each vector pre- and post-sorting.

Credit for this lab: (1) After working diligently on the above, sign up on the signup sheet. There is nothing to be handed in for this last lab, ***but you should spend some additional time to study and understand these algorithms in preparation for the final exam.***