1. Objective:

Implement the sorting algorithms using C++ STL container and iterators.

2. Requirements:

a. ItemType is integer.

b. For first pass, all items are generated randomly and stored in a container and saved into file for the usage of second pass. For the second pass, all items are read in from data file created during the first pass.

c. A container can be vector, deque or list. So, to ensure your program will work on any container, you must use iterators to implement all sorting algorithms. Note that there are different features among three containers such that one of them do not allow removing from the middle. Please read STL API linked from course web site.

d. This program is designed to execute each algorithm for several runs and compute average computing time.

e. Record average computing times (in seconds) for each algorithm on various number of items for first and second pass. To do this, you need to modify the given program on each desired number of items. The given program was designed for the first pass without saving the sorted data into a file. You need to add that part. Other than that addition, you should not need to change any other functions.

f. Implement all sorting algorithm functions. You may add more private or protected functions.

g. You should record the average time of the following and attach at your document:

First pass (Random data)

<table>
<thead>
<tr>
<th>number of items :</th>
<th>10K</th>
<th>50K</th>
<th>100K</th>
<th>500K</th>
<th>1M</th>
<th>2M</th>
<th>4M</th>
</tr>
</thead>
<tbody>
<tr>
<td>Selection Sort</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Insertion Sort</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Bubble Sort</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Merge Sort</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Quick Sort</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Counting Sort</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Radix Sort</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

Second pass (Sorted data)

<table>
<thead>
<tr>
<th>number of items :</th>
<th>10K</th>
<th>50K</th>
<th>100K</th>
<th>500K</th>
<th>1M</th>
<th>2M</th>
<th>4M</th>
</tr>
</thead>
<tbody>
<tr>
<td>Selection Sort</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Insertion Sort</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Bubble Sort</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Merge Sort</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Quick Sort</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Counting Sort</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>
Radix Sort X X X X X X X

Notes: Some algorithms may take long time to complete. Some algorithms may have average run time = 0, i.e. less than 1 sec, K = 1000 and M = 1000000

3. Sample Run

csc313@thecity:~/PROJ/pj3$ ./a.out 1000
Number of items: 1000
*First pass
Selection Sort : Average time : 15
Merge Sort : Average time : 2
Quick Sort : Average time : 0
Counting Sort : Average time : 10
Radix Sort : Average time : 1

*Second pass
Selection Sort : Average time : X
Merge Sort : Average time : X
Quick Sort : Average time : X
Counting Sort : Average time : X
Radix Sort : Average time : X

Extra credit
1. Make your program run for third pass. For the third pass, you manipulate data to work best for specific sorting algorithm and compare the performance with others. For example, counting sort works better when the data value changes in limited ranges. Quick sort works best if data is evenly portioned for next level. You need to generate the data file using a program and submit the program as well as the performance comparison.
2. You may use STL algorithm and compare with your implementation using same data set.

4. Discussion & Documentation

This project has special importance in discussion of various sorting algorithms and their performance. In your documentation, you are required to add good discussion in your documentation. You may add discussion about STL iterator usages in the discussion then remove the other documentation requirements such as architecture design, flow diagram, etc.

<Sort.skel>

//------------------------------
// Project #3 Due Date : Mid-night 11/17/06 Friday
// (Due to the ThanksGiving Recess, you may submit by midnight 11/24/06 without late penalty.
// Objective : Implement the sorting algorithms using C++ STL container and iterators.
// Requirements :
//
// o ItemType is integer.
// o All items are generated randomly and stored in a container
// o A container can be vector, deque or list. So, to ensure
// your program will work on any container, you must
// use iterators to implement all sorting algorithms
//
// o This program is designed to execute each algorithm for
// several runs and compute average computing time
// o Record average computing times (in seconds) for each
// algorithm on various number of items. To do this,
// you need to run this program on each desired number of items.
// o You should not need to change any functions, just implement
// those sorting algorithm functions. You may add more private
// or protected functions
// o You should record the average time of the following:
//
//  number of items : 10K 50K 100K 500K 1M 2M 4M
//  -----------------------------------------------------
//  Selection Sort X X X - - - -
//  Insertion Sort X X X - - - -
//  Bubble Sort X X X - - - -
//  Merge Sort X X X X X X X X
//  Quick Sort X X X X X X X X
//  Counting Sort X X X X X X X X
//  Radix Sort X X X X X X X X
//
//  Notes: Some algorithms may take long time to complete
//  Some algorithms may have average run time = 0,
//  i.e. less than 1 sec
//  K = 1000 and M = 1000000
//
//
#include <ctime>
#include <iostream>
#include <iomanip>
#include <vector>
#include <list>
#include <deque>
#include <sys/time.h>
using namespace std;

// Type define for item type and containers.
// You program should work for vector, deque and list.
// To change from one container to another, you should not need
// to modify your program, just need to recompile it
typedef int ItemType;
//typedef vector<ItemType> ADT;
typedef list<ItemType> ADT;
//typedef deque<ItemType> ADT;

// Timer class is used to record execute time of a sorting algorithm
// Do not modify any part of this class.
//
// Example usage :
//
// Timer t;
// t.start() record starting time
// t.stop() record finishing time
// t.interval() return (finishing time - starting time)
// in secs. Note : you may get return value 0
//
class Timer {

public:

    void start()
    {
        time(&sTime);
    }

    void stop()
    {
        time(&eTime);
    }

    int interval() const
    {
        return eTime-sTime;
    }

private:
    // timeval startTime;
    // timeval stopTime;
    time_t sTime, eTime;
};
// This is the main sorting algorithm class. You need to implement
// all sorting algorithms. For your reference, I have completed
// selectionSort().
//
// The following important functions are completed:
//
// o generateRandomData() : for generating random items and storing
//    in "items" container
//
// o checkResult() : for checking final ordering in "items". It will
//    terminate this program if the result is incorrect.
//
// o displayItems() : display items in "items". Good for debugging.
//
// o swap() : swap two items in "items"
//
// o run() : for executing specific sorting algorithm
//    It measures average execution time
//
// o runall() : execute all sorting algorithms with specific number
//    of items
//
// o Sort(int) : Constructor. To specify the number of items only.
//    default - items range = [0,500000); number of runs = 5
//
// o Sort(int itemSize, int numItems, int numRuns : Constructor.
//    To specify number of items, items range, and number of runs
//
// Notes : "items" is the container
//
//------------------------------------------------------------------

class Sort {

public:

    // see above explanation
    Sort(int numItems);
    Sort(int itemSize, int numItems, int numRuns);

    // all run() several times for different sorting functions
    void runall();

protected:

    // run any sorting algorithm
    // use function pointer to point to a sorting function
    void run(void(Sort::*func)(void));
// entry point for each sorting algorithm
void selectionSort(void);
void insertionSort(void);
void bubbleSort(void);
void mergeSort(void);
void quickSort(void);
void radixSort(void);
void countingSort(void);

private:

// add private functions here....
// example : for mergeSort(), i have two additional functions :
//
// void _merge(ADT::iterator first, ADT::iterator mid, ADT::iterator last);
// void _mergeSort(ADT::iterator first, ADT::iterator last);

void _merge(ADT::iterator first, ADT::iterator mid, ADT::iterator last);
void _mergeSort(ADT::iterator first, ADT::iterator last);

ADT::iterator _partition(ADT::iterator first, ADT::iterator last);
void _quickSort(ADT::iterator first, ADT::iterator last);

int countDigit(void);
int extractDigit(ItemType item, const int pos);
void coutingSortDigit(const int pos);

// see above explanation
void generateRandomData();
void displayItems();
void checkResult();
void swap(ItemType& item1, ItemType& item2);

// Data members :

int maxitem; // items range = [0,maxitem)
int numruns; // number of runs on each sort
int size;   // number of randomly generated items
ADT items; // container that holds items

};

// constructor
Sort::Sort(int numItems):maxitem(100000),numruns(5),size(numItems)
// constructor
Sort::Sort(int itemSize, int numItems, int numRuns):
    maxitem(itemSize), numruns(numRuns), size(numItems)
{
}

// execute all sorting algorithms by calling run() function.
// for largest sizes, you can block selectionSort, insertionSort
// and bubbleSort here.
// Note : How a function can be passed as parameter

void Sort::runall()
{
    cout << "Number of items: " << size << endl;
    cout << "Selection Sort : "; run(&Sort::selectionSort);
    // cout << "Insertion Sort : "; run(&Sort::insertionSort);
    // cout << "Bubble Sort   : "; run(&Sort::bubbleSort);
    cout << "Merge Sort   : "; run(&Sort::mergeSort);
    cout << "Quick Sort   : "; run(&Sort::quickSort);
    cout << "Counting Sort : "; run(&Sort::countingSort);
    cout << "Radix Sort   : "; run(&Sort::radixSort);
}

// run a sorting algorithm
// function pointer points to a sorting function
void Sort::run(void(Sort::*func)(void))
{
    Timer timer;
    int totaltime;

    totaltime = 0;

    // repeat for several runs
    for (int i=1; i<=numruns; i++) {

        // generate random data
generateRandomData();

        if ((i==1) && (size <= 10)) displayItems();
        // time and execute sorting function
timer.start(); (this->*func)(); timer.stop();
        if ((i==1) && (size <= 10)) displayItems();

        // check result
        checkResult();

        // update time
        totaltime += timer.interval();
// print average time
//cout << "Average time : " << setprecision(2)
cout << "Average time : " << totaltime/numruns << endl;
}

// generate random data
void Sort::generateRandomData()
{
    // assume size >= 1
    // You need to implement here
}

// print all items. Only useful if size is small
void Sort::displayItems()
{
    // for displaying items
    // use to debug
    ADT::iterator u;

    u = items.begin();
    for (int i=1; i <= size; i++)
        cout << *u++ << " | ";
    cout << endl;
}

// check sorted container
// terminate program if the result is not correct
void Sort::checkResult()
{
    ADT::iterator u,v;

    u = items.begin();

    // assume size >= 1
    for (int i=1; i < size; i++) {
        v=u++;

        // if out of order, terminate program now!!
        if (*v > *u) {
            cout << "Error at position " << i << " | "
                 << *v << " | " << *u << endl;

            // may add more debug statements
            exit(1);
        }
    }
}
// swap 2 items
void Sort::swap(ItemType& item1, ItemType& item2)
{
    // You need to implement here
}

// Selection Sort
// Only minor modification from the class notes
void Sort::selectionSort(void)
{
    // You need to implement here
}

void Sort::insertionSort(void)
{
    // You need to implement here
}

void Sort::bubbleSort(void)
{
    // You need to implement here
}

// merger [first,mid) and [mid,last)
void Sort::_merge(ADT::iterator first, ADT::iterator mid, ADT::iterator last)
{
    // You need to implement here
}

// consider range [first,last)
void Sort::_mergeSort(ADT::iterator first, ADT::iterator last)
{
    // You need to implement here
}
void mergeSort(void)
{
    _mergeSort(items.begin(), items.end());
}

// consider range [first,last) and >1 items
ADT::iterator _partition(ADT::iterator first, ADT::iterator last)
{
    // You need to implement here
}

void _quickSort(ADT::iterator first, ADT::iterator last)
{
    // You need to implement here
}

void quickSort(void)
{
    _quickSort(items.begin(), items.end());
}

// assume max item=100...000
int countDigit(void)
{
    // You need to implement here
}

int extractDigit(ItemType item, const int pos)
{
    // You need to implement here
}

void countingSortDigit(const int pos)
{
    // You need to implement here
}

void radixSort(void)
{
// You need to implement here
}

void Sort::countingSort(void)
{
    // You need to implement here
}

int main(int argc, char *argv[])
{
    
    // run with size = 10000
    if (argc == 1) {
        Sort s(10000);
        s.runall();
    }
    
    // run with size from argv[1]
    else if (argc == 2)
    {
        Sort s(atoi(argv[1]));
        s.runall();
    }
    else
    {
        // handle other cases here
    }
}