Assignment 4

Due May 14, 2015 by 11:59pm **Points** 100 **Submitting** a text entry box or a file upload
Available after May 7, 2015 at 4pm

Sorting - Mergesort

Purpose

Although merge sort shows the same execution complexity as quick sort, (i.e., *O(n log n)*), its practical performance is much slower than quick sort due to array copying operations at each recursive call. This programming assignments improves the performance of merge sort by implementing a nonrecursive, semi-in-place version of the merge-sort algorithm.

In-Place Sorting

In-place sorting is to sort data items without using additional arrays. For instance, quicksort performs partitioning operations by simply repeating a swapping operation on two data items in a given array, which thus needs no extra arrays.

On the other hand, mergesort we have studied allocates a temporary array, sorts partial data items in that array, and copies them back to the orignal array at each recursive call. Due to this repetitive array-copying operations, mergesort is much slower than quicksort although their running time is upper-bounded to *O(n log n)*.

It has been an research interest to develop in-place mergesort algorithms, almost all of which are however inpractically complicated. Yet, we can improve the performance of mergesort with adding the following two restrictions:

1. Using a nonrecursive method
2. Using only one additional array, (i.e., a semi-in-place method). Merge data from the orignal into the additional array at the very bottom stage, and thereafter perform the next merge from the addtional into the orignal array, in which way merge operations are applied to the original and additional array in turn as you go through each repetitive stage. Don't copy back intermadiate results to the orignal array at the end of each stage for the purpose of always merging data from the orignal to the additional array.

Note that we still need to allow data items to be copied between the original and this additional arrays as many times as *O(log n)*.

Statement of Work

1. Design and implement a nonrecursive, semi-in-place version of the merge-sort algorithm. The framework of your mergesort function will be

```
#include <vector>
#include <math.h> // may need to use pow( )
using namespace std;

template <class Comparable>
void mergesortImproved( vector<Comparable> &a ) {

  int size = a.size( );
  vector<Comparable> b( size );  // this is only one temporary array.

  // implement a nonrecursive mergesort only using vectors a and b.
}
```

Needless to say, the above *mergesortImproved( )* function must not call itself or some other recursive functions. Furthermore, the algorithm should still be based on the same divide-and-conquer approach.

2. Use the following driver function to verify and evaluate the performance of your nonrecursive, semi-in-place mergesort program. The code below assumes that your program is written in the "mergesortImproved.cpp" file.

```
#include <iostream>
#include <vector>
#include <stdlib.h>
#include <sys/time.h>
#include "mergesortImproved.cpp"                    // implement your mergesort
using namespace std;


// array initialization with random numbers
void initArray( vector<int> &array, int randMax ) {
  int size = array.size( );

  for ( int i = 0; i < size; ) {
    int tmp = ( randMax == -1 ) ? rand( ) : rand( ) % randMax;
    bool hit = false;
    for ( int j = 0; j < i; j++ ) {
      if ( array[j] == tmp ) {
        hit = true;
        break;
      }
    }
    if ( hit )
      continue;
    array[i] = tmp;
    i++;
  }
}

// array printing
void printArray( vector<int> &array, char arrayName[] ) {
  int size = array.size( );
```

```
  for ( int i = 0; i < size; i++ )
    cout << arrayName << "[" << i << "] = " << array[i] << endl;
}

// performance evaluation
int elapsed( timeval &startTime, timeval &endTime ) {
  return ( endTime.tv_sec - startTime.tv_sec ) * 1000000
    + ( endTime.tv_usec - startTime.tv_usec );
}

int main( int argc, char* argv[] ) {
  // verify arguments

  if ( argc != 2 ) {
    cerr << "usage: a.out size" << endl;
    return -1;
  }

  // verify an array size

  int size = atoi( argv[1] );
  if ( size <= 0 ) {
    cerr << "array size must be positive" << endl;
    return -1;
  }

  // array generation

  srand( 1 );
  vector<int> items( size );
  initArray( items, size );
  cout << "initial:" << endl;   // comment out when evaluating performance only
  printArray( items, "items" ); // comment out when evaluating performance only

  // mergesort
  struct timeval startTime, endTime;
  gettimeofday( &startTime, 0 );
  mergesortImproved( items );
  gettimeofday( &endTime, 0 );
  cout << "elapsed time: " << elapsed( startTime, endTime ) << endl;

  cout << "sorted:" << endl;    // comment out when evaluating performance only
  printArray( items, "items" ); // comment out when evaluating performance only

  return 0;
}
```

3. Obtain the usual mergsort and quicksort from **Lecture Slides** page, or from your textbook code.

4. Compare the performance among the usual quicksort, the usual mergesort, and your improved mergesort as increasing the array size = 10, 100, 1000, and 10000.

What to Turn in

Clearly state in your code comments any other assumptions you have made. Turn in:

(1) your nonrecursive, semi-in-place mergesort program, (i.e., template <class Comparable> void mergesortImproved(vector<Comparable> &a) in "mergesortImproved.cpp".) (**Don't use different function name or file name!**), and

(2) a separate report in .doc or .docx that must includes:

(2a) a one-page output of your improved mergesort program (when #items = 30), and

(2b) a graph that compares the performance among the usual quicksort, the useual mergesort, and your improved mergesort.

Grading Guide and Answers

Check the following grading guide to see how your homework will be graded. Key answer will be made available after the due date through **Solution** page.

```
Program 4 Grade Guideline

1. Documentation (20pts)
   One page output (a.out 30 will fit one page.)
   Correct(10pts)           1 ~ 2 errors(5pt)     3+ errors or no results(0pt)

   Performance comparison between your algorithm and ordinary merge/quick sorts
   Your algorithm worked faster as increasing the array size (10pts)
   Little difference between your algorithm and others       (5pt)
   Unsatisfactory comparison                                 (0pt)

2. Correctness (60 pts)

  Compiled(20 pt)  Compilation errors(0pt) : If not compiled, 0 for this correctness category

   No recursion
   Never called the same function (10pts)     Called the same function    (0pt)

   One additional array besides the original array passed from main( )
   Declared only one additional   (10pts)     Declared two or more arrays (0pt)
   array

   One way assignment from one to another array in each iteration
   Yes(10pts)                       No, data assigned back and forth
                                             between two arrays in each
```

iteration! (0pt)

Your algorithm is still based on the same divide-and-conquer approach
Yes (10pts) No, totally different (0pt)

3. Program Organization (20pts)
   You must write a plenty of comments to help the professor or the grader understand your code.

   Proper comments
   Good (10pts) Fair (7pts) Poor(3pts) No explanations(0pt)

   Coding style (proper identations, blank lines, variable names, and non-redundant code)
   Good (10pts) Fair (7pts) Poor(3pts) No explanations(0pt)