# DSA/CS 5005 – Computing Structures – Programming Project – 3 – Fall 2016
# (Due: December 9, 2016 11:59 PM)

## Objectives

1. [5 pts] Use the PRNG to generate random numbers with current time as the seed.
2. [5 pts] Implement gen_data() method
3. [10 pts] Implement Bubble sort method
4. [10 pts] Implement Quick sort method
5. [10 pts] Implement Merge sort method
6. [50 pts] Design and implementation
   a. [5 pts] Generate fake survey data
   b. [30 pts] Perform table sorting using different algorithms (10 pts each)
   c. [ 5 pts] Release the memory allocated for the data structures (check main())
   d. [10 pts] Run the program with 100, 500, 1000, 2000, 3000 rows. Record the execution time, then draw a 2D line chart (use Excel or any program you like). Each line represents a different algorithm, X-axis is the number of rows and Y-axis is the execution time. Which one is fastest/slowest? Why? Put the chart and your explanation in a PDF (or Word) document and submit it with your program.
7. [10 pts] Document your project thoroughly as the examples in the textbook. This includes, but is not limited to, header comments for all classes/methods, explanatory comments for each section of code, meaningful variable and method names, and consistent indentation.
8. [10 pts] Bonus: use "function pointer" to select the sorting algorithm that will be used in table-sorting function.

Note: If you are not able to complete all the objectives, show the portions that you have completed. For example, you can get 10 points for objective 3 if you (1) implement Bubble sort, then (2) use it to sort an array of integers and (3) show the array before and after sorting it.

## Discussion

Statistical surveys are one of the research methods in many areas such as marketing and social science. Sorting algorithms are used in the software to help data analysis. In this Project, you will (1) use PRNG to generate fake survey data and (2) implement and evaluate different sorting algorithms before the survey results are available.

### Pseudo Random Number Generator (PRNG)

The pseudo random number generator `(int rand(void))` built in C/C++ returns a pseudo-random integer number between 0 and RAND_MAX (a constant). If you want to generate a random number between 0 and 99, you can get it by using `rand()%100`. If you keep calling this function/method it will get you a sequence of pseudo random numbers. However, the PRNG returns the same sequence each time you run the program if you do not give it a different seed by calling `srand()`. A common choice of the seed

is the current time. You can check http://www.cplusplus.com/reference/clibrary/cstdlib/rand/ for the example of how to initialize the seed using current time and generate random numbers.

### Generate (Fake) Survey Data

Assume we are interested in the ages, genders, zip codes and weights of the subjects in the survey (and ignore other information for the time being). For simplicity, the data will be stored in a 2 dimensional array of integers as follows:

| Row | Age | Gender | Zip Code | Weight |
|---|---|---|---|---|
| 0 | 32 | 1 | 73021 | 150 |
| 1 | 23 | 1 | 73069 | 122 |
| 2 | 32 | 1 | 73072 | 131 |
| 3 | 23 | 0 | 73019 | 142 |
| 4 | 42 | 0 | 73072 | 123 |

You can use the following code to dynamically create a dimensional array of N rows and M columns:

```
int** data;

// allocate memory
data = new int*[M];  // M rows
for( int i=0; i< M; i++)
   data[i] = new int[N];  // for each row, there are N columns

// release memory after use
for( int i=0; i< M; i++)  // delete each row
   delete [] data[i];
delete [] data;
```

The size of this 2 dimensional array is N x 4 in this project, where N is the number of surveys collected. You will write a method to generate a large number of copies of survey data using PRNG and store them in a dynamically allocated array. The prototype of this method is:

```
int** gen_data( int N);
```

This method will create a 2 dimensional array if the size is N x 4. Then populate data into these “surveys” using the following algorithm:

```
1.  Create an N x 4 array (data)
2.  initiate the seed for the PRNG using time()
3.  for i = 0 to N-1
4.     data[i][0] = random_num (20, 60)
5.     data[i][1] = random_num( 0, 1);
6.     data[i][2] = random_num( 73000, 74999);
7.     data[i][3] = random_num(100, 200 );
8.  end for
9.  return data
```

Where `random_num(x, y)` returns an integer between x and y (x < y). We use 0 to represent female and 1 for male in gender.

## Table Sorting

Table sorting is a common feature in statistical analysis or spreadsheet applications. A grid of cells is used to organize data in these applications. What table sorting usually does is sort the rows by the value of some desired column. For example, the data from the survey can be represented as the following table. (Note: Orig. Row is not part of the data)

Table 1

| Orig. Row | Age | Gender | Zip Code | Weight |
|---|---|---|---|---|
| 0 | 32 | 1 | 73021 | 150 |
| 1 | 23 | 1 | 73069 | 122 |
| 2 | 32 | 1 | 73072 | 131 |
| 3 | 23 | 0 | 73019 | 142 |
| 4 | 42 | 0 | 73072 | 123 |

If the table is sorted by “Age”, the new table will be:

Table 2

| Orig. Row | **Age** | Gender | Zip Code | Weight |
|---|---|---|---|---|
| 1 | **23** | 1 | 73069 | 122 |
| 3 | **23** | 0 | 73019 | 142 |
| 0 | **32** | 1 | 73021 | 150 |
| 2 | **32** | 1 | 73072 | 131 |
| 4 | **42** | 0 | 73072 | 123 |

The table also can be sorted by multiple columns in a given sequence. For example, by “Age”, “Gender” then “Zip Code”. Since Table 2 is already sorted by “Age”, next is to sort by “Gender” for each “Age” value.

Table 3

| Orig. Row | **Age** | **Gender** | Zip Code | Weight |
|---|---|---|---|---|
| 3 | **23** | **1** | 73019 | 142 |
| 1 | **23** | **0** | 73069 | 122 |
| 0 | **32** | **1** | 73021 | 150 |
| 2 | **32** | **1** | 73072 | 131 |
| 4 | **42** | **0** | 73072 | 123 |

Then sort each (Age, Gender) by “Zip Code”:

Table 4

| Orig. Row | **Age** | **Gender** | **Zip Code** | Weight |
|---|---|---|---|---|
| 3 | **23** | **1** | **73019** | |
| 1 | **23** | **0** | **73069** | |
| 2 | **32** | **1** | **73072** | |
| 0 | **32** | **1** | **73021** | |
| 4 | **42** | **0** | **73072** | |

***Implementation***

Moving the rows in a large table is costly. Hence *we will create an array to store the indices of the rows. Instead of moving the rows during sorting, the indices will be moved in this array*. The content of this array is shown in the "Orig. Row" column in previous example.

The prototype of a generic table sorting method for this project looks like this:

```
int* GenericTableSorting( int** data, int row, int col,
                          int* seq, int seq_size);
```

where

data: the 2 dimensional array
row/col: size of data
seq: an array states the order of the columns being used in table sorting
seq_size: size of seq

It returns an array of integers, which represents the table after sorting.

For the previous example, data is shown in the following table, row is 5, col is 4, seq[] is { 1, 2, 3} and seq_size is 3. It returns a pointer to an array with content {3, 1, 2, 0, 4}.

| | | | |
|---|---|---|---|
| 32 | 1 | 73021 | 150 |
| 23 | 1 | 73069 | 122 |
| 32 | 1 | 73072 | 131 |
| 23 | 0 | 73019 | 142 |
| 42 | 0 | 73072 | 123 |

The following algorithm can be used to implement GenericTableSorting().

```
1. Create an array (result) of size row, the value of each cell is its index
2. Sort result using the value of seq[0] column in data
3. for j = 1 to seq_size -1
4.    for i = 0 to row -1
5.       begin = i, end = begin;
6.       while( values of columns seq[0] through seq[j-1] in data
7.              of result[end] are the same as of result[begin] )
8.       i = i+1, end = i;
9.      end while
10.     sort result using the value of seq[j] column in data from begin to end-1
11.     i = end;
12.   end for
13. end for
14. return result
```

The idea of this algorithm is quite simple, the table (**result**) is sorted by the column **seq[0]**. Then for each of the remaining **seq[j]**, it scans from the beginning and finds the rows with the same values for columns **seq[0]** through **seq[j-1]** and sorts these rows.

You may create other functions to simplify the implementation. For example, you can replace line 10 by creating the following method:

```
void GenericSort( int** data, int row, int col,
                  int* result,
                  int begin, int end, int target_col);
```

which sorts **result** for the rows between **result[begin]** and **result[end]** according to the values of column **target_col**.

In this project, you will implement 3 different table sorting methods: BubbleTableSorting(), QuickTableSorting() and MergeTableSorting() and use BubbleSort(), QuickSort() and MergeSort() in them respectively.

***Function Pointer [10% bonus]***

Function pointers are pointers to the address of functions and hence can be passed to other functions as arguments. The beauty of function pointers is you can use them to select which function you want to use in another function. For example, you need to implement BubbleTableSorting(), QuickTableSorting() and MergeTableSorting() that have the same code except the sorting algorithms. By using function pointer, you only need to implement a generic table-sorting function and pass the sorting algorithm of your choice to it. Learn how to use function pointers and use function pointers to implement GenericTableSorting(). You will receive 10 extra pts in this project by doing this.

Note: You will need to change the prototype of GenericTableSorting() since one additional parameter (function pointer) will be passed to it.

## main()

main() for this project is provided on learn.ou.edu, it does the following tasks:

(1) Generate (fake) survey data using gen_data().
(2) Perform table sorting using different sorting algorithms.
(3) Print out the execution time using different sorting algorithms.
(4) Print out the results if there are less than 20 rows of data.
(5) Clean up memory allocated (you need to implement this).

```
// Please do not modify it unless
// (1) some of your functions do not work and you are commenting out them to make the
//     program partially work. OR
// (2) You did not finish some of the objectives but you want to demonstrate the part
//     you have finished

// headers
#include <windows.h>
#include <iostream>
#include <cstdio>
#include <cstdlib>
#include <ctime>

// your headers

using namespace std;


// your code
```

```
int main(int argc, char* argv[])
{
   int** data;
   int rows;
   int* seq;
   int seq_size;
   int* bubble_result;
   int* quick_result;
   int* merge_result;

   LARGE_INTEGER start[3], stop[3]; //

   if( argc!= 2)
   {
        cout << "Usage: project5.exe [rows]" << endl;
        exit(0);
   }

   rows = atoi( argv[1]);

   // generate data
   data = gen_data( rows);

   if( rows <= 20) // print out the raw data for small data set
   {
        cout << "Raw data: " << endl;
        cout << "====================================" << endl;
        for( int i=0; i< 20; i++)
               cout << i << ")\t"<< data[i][0] << " " << data[i][1] << " "
                                  << data[i][2] << " " << data[i][3] << endl;
        cout << "====================================" << endl;
   }


   // generate the sequence of columns for table sorting

   if( rows <= 20)
   {
        seq_size = 2;
        seq = new int[seq_size];
        seq[0] = 1; // by gender first
        seq[1] = 0; // then by age
   }
   else
   {
        seq_size = 4;
        seq = new int[seq_size];
        seq[0] = 2; // by zip code first
        seq[1] = 0; // then by age
        seq[2] = 1; // then by gender
        seq[3] = 3; //then by weight
   }

   // Do table sorting and measure the time it takes
   // using different sorting algorithms
   // If you are using function pointers, you need to modify
   // the table sorting code below.
   QueryPerformanceCounter(&start[0]);
   bubble_result = BubbleTableSorting( data, rows, 4, seq, seq_size);
   QueryPerformanceCounter(&stop[0]);

   QueryPerformanceCounter(&start[1]);
   quick_result =QuickTableSorting( data, rows, 4, seq, seq_size);
   QueryPerformanceCounter(&stop[1]);

   QueryPerformanceCounter(&start[2]);
   merge_result = MergeTableSorting( data, rows, 4, seq, seq_size);
   QueryPerformanceCounter(&stop[2]);



     if( rows <= 20) // print out sorted data for small data set
```

```
    {
        cout << "Table Sorting Result: " << endl;
         for( int i=0; i< rows; i++)
             cout << bubble_result[i] << " ";
         cout << endl;
         for( int i=0; i< rows; i++)
             cout << quick_result[i] << " ";
         cout << endl;
         for( int i=0; i< rows; i++)
             cout << merge_result[i] << " ";
         cout << endl;
        cout << "=============================================================================" << endl;
        cout << "Bubble\t\t\tQuick\t\t\tMerge" << endl;
        cout << "=============================================================================" << endl;
         for( int i=0; i< 20; i++)
                cout << data[bubble_result[i]][0] << " " << data[bubble_result[i]][1] << " "
                     << data[bubble_result[i]][2] << " " << data[bubble_result[i]][3] << "\t\t"
                     << data[quick_result[i]][0] << " " << data[quick_result[i]][1] << " "
                     << data[quick_result[i]][2] << " " << data[quick_result[i]][3] << "\t\t"
                     << data[merge_result[i]][0] << " " << data[merge_result[i]][1] << " "
                     << data[merge_result[i]][2] << " " << data[merge_result[i]][3] << endl;
        cout << "=============================================================================" << endl;
    }

       // print out the time taken
    cout << "Bubble\t\t\tQuick\t\t\tMerge" << endl;
    for( int i=0; i < 3; i++)
        cout << stop[i].QuadPart - start[i].QuadPart << "\t\t\t";
    cout << "(time)" << endl;


    // Insert your code here to
    // release memory allocated for
      // data, bubble_result, quick_result, merge_result and seq

    return 0;
}
```

## Constraints

1. In this project, the header files you can use are: `<iostream>`, `<cstring>`, `<cstdlib>`, `<cstdio>`, `<ctime>`, and `<windows.h>`.
2. None of the projects will be a group project. Consulting with other members of this class on programming projects is strictly not allowed and plagiarism charges will be imposed on students who do not follow this.