Programming Language Lab. 5 (A 반)

5.1 Test driver program to measure the elapsed time of sorting functions.

In order to test a function as a module in a large scale system, a driver program is temporarily prepared and used. This driver program prepares the testing conditions (i.e., arrays to be sorted, input data file and output data files, etc.), and invokes the module to be tested.

The measurement of the elapsed time taken to process some function, is usually performed with system clock readings. In Visual C++, QueryPerformanceCounter (LARGE_INTEGER & time) provides the current time in microsecond. So, reading the system time before and after the function call can provide the time taken for the function call. Example of time measurement is as follows:

```
#include <Window.h>
. . . .
int selectionSort(int array[], int size);
//int quickSort(int array[], int size);
//int bubbleSort(int array[], int size);
//int mergeSort(int array[], int size);
// QueryPerformanceFrequency(LARGER_INTEGER *freq);
// QueryPerformanceCounter(LARGER_INTEGER *time);

int main()
{
    LARGE_INTEGER freq, t_1, t_2, t_diff;
    double elapsed_time;
    CONST int MAX_SIZE 1000;
    int array[MAX_SIZE], size;

    . . . .
    //Initialize array with “size” data elements
    // generated by rand()function
    QueryPerformanceFrequency(&freq);
    QueryPerformanceCounter(&t_1);
    selectionSort(array, size);
    QueryPerformanceCounter(&t_2);
    t_diff = t_2.QuadPart - t_1.QuadPart;
    elapsed_time = ((double) t_diff / freq.QuadPart)*1000000;
       // in microsec
    cout << “It took ” << elapsed_time << “micro-seconds to sort ”
         << size << “ integer data array.” << endl;
    . . . .

}
```

Write a selectionSort() function that receives parameters of integer array and its size, and sorts the array.

Write a driver program that prepares 4 integer arrays with 100, 1000, 10,000 and 100,000 data elements that have been generated using rand() function. The driver program should check the time taken for the selection sorting of the 4 integer arrays, individually, and print out the elapsed times for each array size.

5.2 Merge Sort

Merge sort is an *O*(*n* log *n*) comparison-based sorting algorithm.

Conceptually, a merge sort works as follows

① If the list is of length 0 or 1, then it is already sorted. Otherwise:
② Divide the unsorted list into two sublists of about half the size.
③ Sort each sublist recursively by re-applying the merge sort.
④ Merge the two sublists back into one sorted list.

Merge sort incorporates two main ideas to improve its runtime:

① A small list will take fewer steps to sort than a large list.
② Fewer steps are required to construct a sorted list from two sorted lists than two unsorted lists. For example, you only have to traverse each list once if they're already sorted (see the merge function below for an example implementation).

In simple pseudocode, the algorithm might be expressed as this:

```
function mergeSort(int array[], int left, int right, int size)
{
    find middle;
    if (left < right) {
        mergeSort(array, left, middle, size);
        mergeSort(array, middle+1, right, size);
        merge(array, left, middle, right, size);
    }
}
function merge(int array[], int left, int middle, int right, int size)
{
    int temp[MAX_SIZE];
    i = left; j = middle +1; k = 0;
    while (i <= middle && j <= right)
    {
        if (array[i] > array[j])
           temp[k++] = array[j++];
        else
           temp[k++] = array[i++];

        if (i>middle)
         { m = j; n = middle;}
        else
         {m = i; n = middle;}

        for ( ; m<=n; m++)
          temp[k++] = array[m];
        for (m=left; m<= right; m++)
          array[m] = temp[m];
}
```

Write a mergeSort() function, and measure the time taken to sort, using the driver program that has been developed in Lab. 5.1.