ECE 71/191T – Data Structures and Algorithms

Dr. Gregory R. Kriehn, Fresno State

C++ Homework Assignment: Chapter 18

Code Due By: Midnight on Mon, Apr 17
Write-up Due By: Class on Tue, Apr 18

HOMEWORK #34 – Computational Complexity of Sorting Algorithms

Write a program that creates 4 identical arrays, **`list1`**, **`list2`**, **`list3`**, and **`list4`**, and a fifth list implemented as a linked list, **`list5`** – each 10,000 elements/nodes long. Populate the first list with a set of random integers between 1 and 100,000, and copy the numbers into each of the 4 other lists. In other words, start with an identical set of 5 lists of unsorted random integers (4 arrays and 1 linked list), 10,000 elements/nodes long using numbers between 1 – 100,000.

Create a template class called **`searchSortAlgorithms.h`** and implement the following template class functions:

```
// Searching Algorithms
template <class elemType>
int seqSearch(const elemType list[], int length,
     const elemType &item)

template <class elemType>
int binarySearch(const elemType list[], int length,
     const elemType &item)


// Sorting Algorithms
template <class elemType>
void bubbleSort(elemType list[], int length)

template <class elemType>
void selectionSort(elemType list[], int length)

template <class elemType>
void insertionSort(elemType list[], int length)

template <class elemType>
void quickSort(elemType list[], int length)


// Functions that Support the Sorting Algorithms
template <class elemType>
void swap(elemType list[], int first, int second)

template <class elemType>
```

```
int minLocation(elemType list[], int first, int last)

template <class elemType>
void recQuickSort(elemType list[], int first, int last)

template <class elemType>
int partition(elemType list[], int first, int last)
```

Use the code in Malik to guide the development of your template class functions.

NOTE: Because these functions are to be implemented independent of any class object, and because several of the sorting functions are dependent upon the additional template functions `swap()`, `minLocation()`, `recQuickSort()`, and `partition()`, you will have to include function template prototypes in addition to the function template definitions, as necessary.

The final sorting algorithm to be implemented is the merge sort, which operates on a linked list. Therefore, modify your **`unorderedLinkedList.h`** template (recall that the class template is derived from **`linkedList.h`**) and add the following member functions:

```
public:
     void mergeSort();

private:
     void recMergeSort(nodeType<Type>* &head);

     void divideList(nodeType<Type> *first1,
          nodeType<Type>* &first2);

     nodeType<Type> *mergeList(nodeType<Type> *first1,
          nodeType<Type> *first2);
```

The private member functions listed here support the implementation of the merge sort algorithm. Notice that two of the functions accept references to `nodeType<Type>` pointers. Implement the member functions, as necessary, out of Malik.

NOTE: On pg. 1309, Malik has an error within the `divideList()` function. The correct implementation of the `if – else if – else` construct is:

```
…
if (first1 == nullptr)             // list is empty
     first2 = nullptr;
else if (first1->link == nullptr)  // list has only 1 node
     first2->link = nullptr;       // CORRECTION
else
```

```
{
      …
}
```

NOTE: The **`recMergeSort()`** function requires the use of the **`first`** and **`last`** pointers (implemented as private members) found in **`linkedList.h`**. To avoid scope resolution problems, use **`this->first`** and **`this->fast`** to force the compiler to treat **`first`** and **`last`** as dependent names within your template classes.

Carefully verify the operation of your sorting algorithms before proceeding further.

Finally, write a program that compares the time necessary to implement each of the sorting algorithms. Use the **`clock()`** function and the **`CLOCKS_PER_SEC`** constant in **`<ctime>`**, as necessary. Print each of the required execution times for the 5 algorithms to the screen for your 10,000 element unsorted arrays/linked list.

```
Bubble Sorting Time:     3.573e-01 s
Selection Sorting Time:  1.580e-01 s
Insertion Sorting Time:  9.478e-02 s
Quick Sorting Time:      1.568e-03 s
Merge Sorting Time:      1.743e-03 s
```

Typically, the quick sort algorithm is **`O(n log n)`**, but the worst case scenario is **$O(n^2)$**, whereas merge sort is always **`O(n log n)`**. Vary the number of elements from from 1 to 100,000. Does the quick sort algorithm ever perform more poorly than the merge sort algorithm with your lists of random integers? Why or why not?

EXTRA CREDIT: Record the sorting times for a variety of array/linked list sizes varying between 1 – 100,000 for each of the 5 sorting algorithms, and import the data into Matlab. As the element sizes grow, plot the data points and illustrate that the Bubble, Selection, and Insertion sort algorithms grow as **$O(n^2)$** and, and Quick and Merge sort grow as **`O(n log n)`**. Include a plot and an explanation for your write-up, as necessary.