**Data Structures - Chapter 13 - Programming Assignment   
Implementing a Simple Quicksort Function**

**The Assignment (part 1):**

You will implement and test the quicksort function. This simple implementation will sort an array of integers and follow the pseudocode from Chapter 13.

**Purposes:**

Ensure that you can implement the recursive pseudocode for quicksort.

**Before Starting:**

Read all of Chapter 13.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**File that you must write:**

1. quick1.h: Header file for this version of the quicksort function. You don't have to write much of this file. Just copy our version from [quick1.h](http.cs.colorado.edu/pub/techreports/main/programs/quick1.h) and add your name and other information at the top. NOTE: This header file contains only one prototype (for the quicksort function) and its documentation. It does not have any class declarations.
2. quick1.cxx: This file should contain the implementations of the simple quicksort function that is described in the header file.

**The Simple Quicksort Function   
Discussion of the Assignment**

There may be several functions in your quick1.cxx (such as the partition function). But quick1.h must only have ONE prototype (the prototype for the quicksort function). Any other functions that are in quick1.cxx should be declared with "internal linking". This hides the function name from the linker, and is accomplished by placing the word static before the function prototype and before the function definition. For example: static void partition(int data[ ], size\_t n, size\_t& pivot\_index)...

Modify the quick1exam.cpp so it will test at least these tests for grading:

1. Calls quicksort(NULL, 0) to make sure that it doesn't crash.
2. Calls quicksort to sort an array with just one element.
3. Calls quicksort to sort an array with 1000 elements (containing the numbers 0 through 1000 in a random order). See Note 4 below. Check that the array is correct after sorting.
4. Calls quicksort to sort an array that contains 1000 elements with two copies of the numbers from 0 to 499. Check that the array is correct after sorting.
5. Calls quicksort to sort an array that contains the numbers 0 through 25 already sorted.
6. Calls quicksort to sort an array that contains the numbers 25 through 0 in reverse order.

Write the a precondition/postcondition contract for each function that you implement (such as partition). Put these contracts in your .cxx file.

(Part 2) Comparison of sorting algorithms:

1. Generate int arrays with N (N up to 1000000) random integers (where the numbers are from 0 to N). Sort them with Selection Sort algorithm as described in class. Record the timing in each case. Compare with Quick Sort as developed by you. Use the high\_resolution\_clock::time\_point in the <chrono> library for the timing measurements. Also, sum all the elements in the array mod N (where N=the size of the array) to convince yourself that all the elements are still in the array after the sorting (their sums must be the same).
2. Generate an int array of 1000000 elements containing the values of

data[0]=0

data[1]=0

…

data[999999]=999999. Now sort this array with Quick Sort with different pivots:

Compare the performance of Quick Sort using (a) the first element as the pivot (this will give you the worst case performance) and (b) a better choice of a pivot is to take the middle value of 3 randomly chosen elements in the array.

This is the selection sort algorithm:

**void selectionsort(int data[ ], size\_t n)**

**// Library facilities used: algorithm, cstdlib**

**{**

**size\_t i, j, index\_of\_largest;**

**int largest;**

**if (n == 0)**

**return; // No work for an empty array.**

**for (i = n-1; i > 0; --i)**

**{**

**largest = data[0];**

**index\_of\_largest = 0;**

**for (j = 1; j <= i; ++j)**

**{**

**if (data[j] > largest)**

**{**

**largest = data[j];**

**index\_of\_largest = j;**

**}**

**}**

**swap(data[i], data[index\_of\_largest]);**

**}**

**}**

This is how to use the <chrono> library:

**#include "stdafx.h"**

**#include <iostream>**

**#include <chrono>**

**using namespace std;**

**using namespace std::chrono;**

**int main()**

**{**

**high\_resolution\_clock::time\_point t1 = high\_resolution\_clock::now();**

**int i;**

**for (i = 0; i < 1000000; i++)**

**;**

**high\_resolution\_clock::time\_point t2 = high\_resolution\_clock::now();**

**duration<double> time\_span = t2 - t1;**

**std::cout << "It took " << time\_span.count() << " seconds.";**

**std::cout << std::endl;**

**}**

This is how to generate N integers in the range 0 to N:

**#include "stdafx.h"**

**#include <iostream>**

**#include <cstdlib> // For rand and srand**

**#include <ctime> // For the time function**

**int \*getRandomNumbers(int num);**

**using namespace std;**

**int main()**

**{**

**const int N = 1000;**

**int \*intArray = getRandomNumbers(N);**

**for (int i = 0; i < N; i++)**

**cout << intArray[i] << endl;**

**return 0;**

**}**

**int \*getRandomNumbers(int num)**

**{**

**int \*arr = nullptr; // Array to hold the numbers**

**if (num <= 0)**

**return NULL;**

**arr = new int[num];**

**// Seed the random number generator by passing**

**// the return value of time(0) to srand.**

**srand(time(0));**

**// Populate the array with random numbers.**

**for (int count = 0; count < num; count++)**

**arr[count] = rand() % num;**

**return arr;**

**}**