Exercise 1 (15 pts—5 pts/Q):

Consider the following array:

41	7	11	22	17	3	19	5	58	13
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]

Show the content of the array after the fourth iteration of

a. Bubble Sort
b. Selection Sort
c. Insertion Sort

Exercise 2 (10 pts):

Consider the following array (same as in Exercise 1):

41	7	11	22	17	3	19	5	58	13
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]

Show how the values in the array would have to be rearranged in order to satisfy the heap property.

Exercise 3 (15 pts—5 pts/Q):

Consider the following array:

26	24	3	17	25	24	13	60	47	1
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]

Tell which sorting algorithm would produce the following results after four iterations:

a.

1	3	13	17	26	24	24	25	47	60
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]

b.

1	3	13	17	25	24	24	60	47	26
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]

c.

3	17	24	26	25	24	13	60	47	1
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]

Exercise 4 (5 pts):

How many comparisons would be needed to sort an array containing 100 elements using Selection Sort if the original array values were already sorted?

Exercise 5 (10 pts):

How would you modify the Radix Sort algorithm to sort the list in descending order than ascending order?

Exercise 6 (10 pts):

Determine the Big-O measure for QuickSort based on the number of elements moved rather than the number of comparisons.

a. For the best case
b. For the worst case

Exercise 7 (10 pts):

Does Radix Sort return correct sorting results when the input sequence contains negative elements? If yes, please give your reason. If no, please revise LSD Radix Sort algorithm to deal with negative elements and give your pseudocode.

Exercise 8 (10 pts):

Suppose ***n*** is even and array $A = [a_1, a_2, ..., a_n]$ is semi-identical, i.e., $a_1 = a_2 = ... = a_{n/2}$ and $(a_{n/2})+1 = (a_{n/2})+2 = ... = a_n$. For example, $A = [1, 1, 1, 1, 2, 2, 2, 2]$. What is the quicksort's running time in this case? Please explain your reason.

Exercise 9 (5 pts):

Use the Three-Question Method to verify MergeSort algorithm.

Exercise 10 (10 pts):

The *C++ thread* library provides a function that returns the number of threads that the hardware is capable of running. Modify the parallel *MergeSort* so that the user specifies a minimum chunk size. The program should then use the number of threads available and *MAX_ITEMS* to determine the largest chunk size that will produce that many threads. If the computed chunk size is larger than what the user specifies, use that instead. Here is how to get the number of threads available:

unsigned int maxthreads = thread::hardware_concurrency();