Assignment 2 – Data Structures and Algorithms

Deadline:	Tuesday March 5 by 11:59 pm
Type:	Individual Assignment
Weight:	5%

Theory Questions (35 points):

Q1 (4)

2.1.6 Which method runs faster for an array with all keys identical, selection sort or insertion sort? Explain your answer

Q2 (5)

2.1.14 *Dequeue Sort:* Explain how would you sort a deck of cards, with restriction that only allowed operations are to look at the values of top two cards, to exchange the top two cards, and move the top card to the bottom of the deck.

Q3 (5)

2.1.20 What is the *best case* for *shellsort*? Justify your answer.

Q4 (7)

Write a recursive method which accepts two parameters: head of a linked list and a value *k*. Then it prints the kth to the last element of a linked list.
For example, if the linked list content is **a->b->c->d** and *k* is 3 it prints out *b*

Q5 (5)

Assume that we have a bitonic array (as explained in Assignment 1) with bitonic point p. For example this is a bitonic array: 5 23 75 90 83 76 65 15 12 3. Here 90 is called the bitonic point (the point that the decreasing part begins). Assume that you know the index of bitonic point in a bitonic array of size N. Explain a sorting algorithm with complexity O(N) to sort a bitonic array with size N and bitonic point index p in an ascending order. (hint: merging can help here).

Q6 (4)

a) Explain why Merge sort is the most suited for very large inputs (that do not fit inside memory) while quick sort is not as suited. Note that these two sorting techniques have comparable time complexities.
b) Can Merge sort be performed in place? Explain your understanding.

Q7 (5)

Suppose we are given two sequences *A* and *B* of *n* elements, possibly containing duplicates, on which a total order relation is defined. Describe an efficient algorithm for determining if *A* and *B* contain the same set of elements. What is the
running time of this method?

Programming Questions (65 points):

Q1 (10)

Write a recursive Java method that rearranges an array of integer values so that all the even values appear before all the odd values. Test your program.

Q2 (10)

2.2.12 *Sublinear extra space.* Develop a merge implementation that reduces the extra space requirement to max(M, N/M), based on the following idea: Divide the array into N/M blocks of size M (for simplicity in this description, assume that N is a multiple of M). Then, (i) considering the blocks as items with their first key as the sort key, sort them using selection sort; and (ii) run through the array merging the first block with the second, then the second block with the third, and so forth.

Q3 (10)

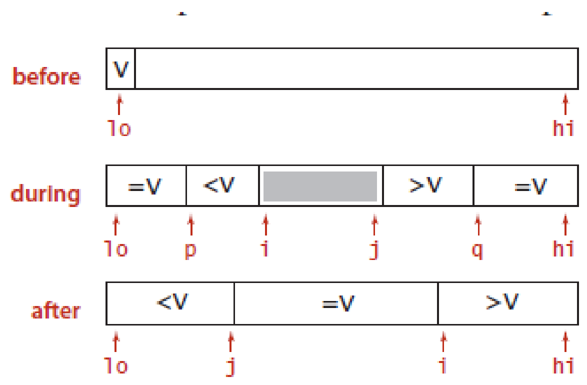
2.2. 20 *Indirect sort.* Develop and implement a version of mergesort that does not re-arrange the array, but returns an int[] array perm such that perm [i] is the index of the i^{th} smallest entry in the array.

Q4 (10)

2.3.12 Show, in the style of the trace given with the code, how the entropy-optimal sort first partitions the array B A B A B A B A C A D A B R A.

Q5: (15)

Fast 3-way partitioning. (J. Bentley and D. Mcllroy) Implement an entropy-optimal sort based on keeping item's with equal keys at both the left and right ends of the subarray. Maintain indices *p* and *q* such that *a[lo..p-1]* and *a[q+1..hi]* are all equal to a[lo], an index *i* such that *a[p..i-1]* are all less than *a[lo]*, and an index *j* such that *a[j+1..q]* are all greater than *a[lo]*. Add to the inner partitioning loop code to swap *a[i]* with *a[p]* (and increment p) if it is equal to *v* and to swap *a[j]* with *a[q]* (and decrement q) if it is equal to *v* before the usual comparisons of *a[i]* and *a[j]* with *v*. After the partitioning loop has terminated, add code to swap the items with equal keys into position.
Note : This code complements the code given in the text, in the sense that it does extra swaps for keys equal to the partitioning item's key, while the code in the text does extra swaps for keys that are *not* equal to the partitioning item's key



Bentley-McIlroy 3-way partitioning

Q6: (10)

Write a java program with the following requirements:

- It defines a class (the class implements Comparable) that represents a Time with 3 attributes; hour, minute and second.
- The program asks the user to enter the size of an array and then creates an array of time objects. The array is filled with a set of time objects. The values (hour, minute, second) of each object can be created randomly or it is entered by the user.
- The program sorts the array of time objects in an ascending order and then shows the results. It should also check if all the time objects are distinct and print a proper message accordingly.