CIS 044:   
Introduction to Data Structures Using Java

Lab 7

**Instructor**

Dr. Maher Mneimneh  
Email:mahernm@gmail.com

**Guidelines**

**Please follow the guidelines below:**

**For a programming question, submit a .java file (for source code) and a .txt file for program output. If the problem involves other questions, submit a separate .txt file to answer the question.**

**P1 (30 points)**

Consider an array of length n that contains unique integers between 1 and n+1. For example, an array A1 of length 5 might contain the integers 3, 6, 5, 1, and 4. In an array of this form, one number between 1 and n+1 will be missing. In A1, the integer 2 is missing from the array. As another example, consider the array A2 that contain the integers 4, 7, 5, 2, 6, and 1. A2 has size 6, contains integers in the range of 1 to 7. In A2, the integer 3 is missing from this array.

Write a Java method

**static int findMissing(int [] a)**

that reads the array and returns the number not present in the array. The running time of the algorithm should be O(n).

For A1=[3,6,5,1,4], **findMissing(A1)** returns 2. For A2=[4,7,5,2,6,1], **findMissing(A2)** returns 3.

**Hint:** Use the formula: 1 + 2 + 3 + …+ n = n(n+1)/2

**P2 (35 points)**

You can sort a large array of m integers that are in the range 1 to n by using an array **count** of n entries to count the number of occurrences of each integer in the array. For example, consider the following array **A** of 14 integers that are in the range from 1 to 9 (note that in this case m = 14 and n = 9):

9 2 4 8 9 4 3 2 8 1 2 7 2 5

Form an array **count** of 9 elements such that **count[i-1]** contains the number of times that **i** occurs in the array to be sorted. Thus, **count** is

1 4 1 2 1 0 1 2 2

In particular, **count[0] = 1** since 1 occurs once in **A**. **count[1] = 4** since 2 occurs 4 times in **A**. **count[2]=1** since 3 occurs once in **A**. **count[3] =2** since 4 occurs 2 times in **A**.

Use the count array to sort the original array **A**. Implement this sorting algorithm in the function

**public static void countingSort(int[] a, int n )**

and analyze its worst case running time in terms of m (the length of array a) and n.

**After calling countingSort(), a must be a sorted array (do not store sorting result in a temporary array).**

**P3 (35 points)**

The median of a collection of values is the middle value. One way to find the median is to sort the data and take the value at the center. But sorting does more than necessary to find the median. You need to find only the kth smallest entry in the collection for an appropriate value of k. To find the median of n items, you would take k as n/2 rounded up

You can use the partitioning strategy of quick sort to find the kth smallest entry in an array. After choosing a pivot and forming the subarrays *Smaller* and *Larger*, you can draw one of the following conclusions:

* If *Smaller* contains k or more entries, it must contain the kth smallest entry
* If *Smaller* contains k-1 entries, the kth smallest entry is the pivot.
* If *Smaller* contains fewer than k-1 entries, the kth smallest entry is in *Larger*.

Implement the recursive method

**public satic int findKSmallest(int[] a, int k)**

that finds the **kth** smallest entry in an unsorted array **a**.

Use the method **findKSmallest** to implement the method

**public static int findMedian(int[] a)**

that finds the median in an array **a**.

**Additional non-graded Problems**

Chapter 9

Project 2 (p. 240), Project 4 (p. 242)

\* Modify insertionSort algorithm so that the loop that shifts elements is replaced by a binary search algorithm. What is the complexity of the resulting algorithm?

\*\* The quick sort algorithm implemented in class requires O(n) extra memory to recursive calls on the stack. Modify quick sort algorithm so that it requires at most O(logn) extra memory.