CS146: Quiz 7
Due Tuesday, March 21, at 7:00AM
10 points

For this quiz you will implement two different versions of the partition function used by QuickSort and you will implement two different versions of CountingSort. **To get started, import the starter files, Partition.java and RadixSort.java into the sorting package you create in a new Java Project.** Please do not change any of the method signatures in either class. Implement the methods described below. You are free to test your code however you prefer. These methods should be completed individually using any IDE you are comfortable with. You are free to use the textbook, slides, class notes, and the Java API Documentation, but **DO NOT** consult any other resources.

Partition.java

`public static int partitionLomuto(int[] arr, int low, int high)`
Implement this method using Lomuto's partition algorithm. This is the algorithm that we discussed in class that chooses the last element as the partition and iterates through the array swapping an element that belongs in the low side with the lowest (farthest left) element on the high side. At the end, the pivot is swapped with the lowest element (farthest left) element on the high side. This method should return the index of the pivot element, which is now in its correct position in the array. See the pseudocode below.

```
partition(arr, p, r)
  x = arr[r]
  i = p - 1
  for j = p to r - 1
    if arr[j] ≤ x
      i = i + 1
      swap arr[i] and arr[j]
  swap arr[i+1] and arr[r]
  return i+1
```

`public static int partitionHoare(int[] arr, int low, int high)`
Hoare's partition algorithm is an alternative form of the partition algorithm. It was actually the algorithm that was used in the original QuickSort implementation. Hoare's algorithm keeps two indices and iterates through the loop from the front and the back simultaneously towards the center. If it finds a pair of elements out of place it swaps them. It terminates when the index keeping track of the low side is greater than or equal to the index keeping track of the high side. In this method, the pivot is not necessarily in the correct location at the end of the partition and *the index returned does not represent the pivot element, but the largest (farthest right) index in the low side.* Every element on the low side is less than or equal to the pivot while every element on the high side is greater than or equal to the pivot. Implement Hoare's partition algorithm using the pseudocode below.

```
paritionHoare(arr,p,r)
  pivot = arr[p]
  i = p-1
  j = r+1
  while true:
    repeat j = j-1 until arr[j]≤pivot
    repeat i = i+1 until arr[i]≥pivot
    if i<j
      swap arr[i] and arr[j]
    else
      return j
```

What sort of input arrays will enable Hoare's algorithm to still create relatively equal size partitions whereas Lumoto's algorithm will create unequal partitions? **Write your answer in the location specified in Partition.java.**

RadixSort.java

RadixSort.java contains two different RadixSort implementations each using a different version of Counting Sort that you will implement. As a reminder the pseudocode for CountingSort discussed in class is as follows.

```
countingSort(arr, n, k)
1  let B[1 : n] and C[0 : k] be new arrays.
2  for i = 0 to k
3    C[i] = 0
4  for j = 1 to n
5    C[arr[j]] = C [arr[j]] + 1
6  for i = 1 to k
7    C[i] = C[i] + C[i – 1]
8  for j = n downto 1
9    B[C[arr[j]]] = arr[j]
10   C[arr[j]] = C[arr[j]] – 1
11  return B
```

`private static void countingSort1(int[] arr, int place)`
For this method you will implement the countingSort algorithm. Note that this implementation is specific to RadixSort and **assumes that k=10** and the `int place` parameter represents which column is currently being sorted (1s, 10s, 100s, etc.). Note that you can use the pseudocode above as a guide, but you will need to think about the correct math to ensure that you are sorting on the right significant digit in the array.

You can test your method independently and using the RadixSort1 function.

`private static void countingSort2(int[] arr, int k)`
Now you implement almost the same method, but instead of iterating from the end of `arr`, you will instead iterate from the beginning. So line 8 becomes **`for j=1 to n`**. You should think about why this alternative does not impact the correctness of CountingSort. **You don't need to write this down, just think about it.**

RadixSort2 is not working correctly even though your countingSort2 is working correctly[1]. Why is RadixSort2 not working? **Write your answer in the location specified in RadixSort.java**

Submission

Please create a jar or zip file of your project. It should include the javas file (Partition.java and RadixSort.java) and submit it on Canvas. If your file includes class files or other extraneous files outside of the package folder and java file, you will receive an automatic one point deduction.

[1] If RadixSort2 is working correctly, make sure that you have implemented the pseudocode as described and did not make any other changes.