To avoid comparing we need to use an algorithm that doesn’t run comparisons on the elements such as radix sort or bucket sort. The provided example compares based on digits, same idea as radix sort. This running time of this sort should run **O(nk),** since we do k stable sots each run O(n) time, k is the number of digits. It’s a linear time sorting algorithm and is more efficient when the range is reasonably small. It’s memory intensive because it’s not in-place but stable though. With this code below, the bottleneck part is the “**countSort**” method because of extracting the digits, and if the number is large, the time with of the execution of this code block is also large. This will cost memory and time if the range is big, but for reasonable ranges the operation shouldn’t cost much and is better than comparison based algorithms.

**class** Problem1Sort {

// A utility function to get maximum value in arr[]

**static** **int** getMax(**int** arr[], **int** n)

{

**int** mx = arr[0];

**for** (**int** i = 1; i < n; i++)

**if** (arr[i] > mx)

mx = arr[i];

**return** mx;

}

// A function to do counting sort of arr[] according to

// the digit represented by exp.

**static** **void** countSort(**int** arr[], **int** n, **int** exp)

{

**int** output[] = **new** **int**[n]; // output array

**int** i;

**int** count[] = **new** **int**[10];

// Arrays.fill(count,0);

// Store count of occurrences in count[]

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

count[ (arr[i]/exp)%10 ]++;

// Change count[i] so that count[i] now contains

// actual position of this digit in output[]

**for** (i = 1; i < 10; i++)

count[i] += count[i - 1];

// Build the output array

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

{

output[count[ (arr[i]/exp)%10 ] - 1] = arr[i];

count[ (arr[i]/exp)%10 ]--;

}

// Copy the output array to arr[], so that arr[] now

// contains sorted numbers according to curent digit

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

arr[i] = output[i];

}

// The main function to that sorts arr[] of size n using

// Radix Sort

**static** **void** Problem1Sort(**int** arr[], **int** n)

{

// Find the maximum number to know number of digits

**int** m = *getMax*(arr, n);

// Do counting sort for every digit. Note that instead

// of passing digit number, exp is passed. exp is 10^i

// where i is current digit number

**for** (**int** exp = 1; m/exp > 0; exp \*= 10)

*countSort*(arr, n, exp);

}

// A utility function to print an array

**static** **void** print(**int** arr[], **int** n)

{

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

System.***out***.print(arr[i]+" ");

}

/\*Driver function to check for above function\*/

**public** **static** **void** main (String[] args)

{

**int** arr[] = {170, 45, 75, 90, 802, 24, 66, 66};

**int** n = arr.length;

*Problem1Sort*(arr, n);

*print*(arr, n);

}

}

d) yes, it is stable. Because the same key appear in the output array in the same order as they do in the input array. That is, it breaks ties between two elements by the rule that whichever element appears first in the input array appears first in the output array, from the input array in the code above we can infer stability.

e)