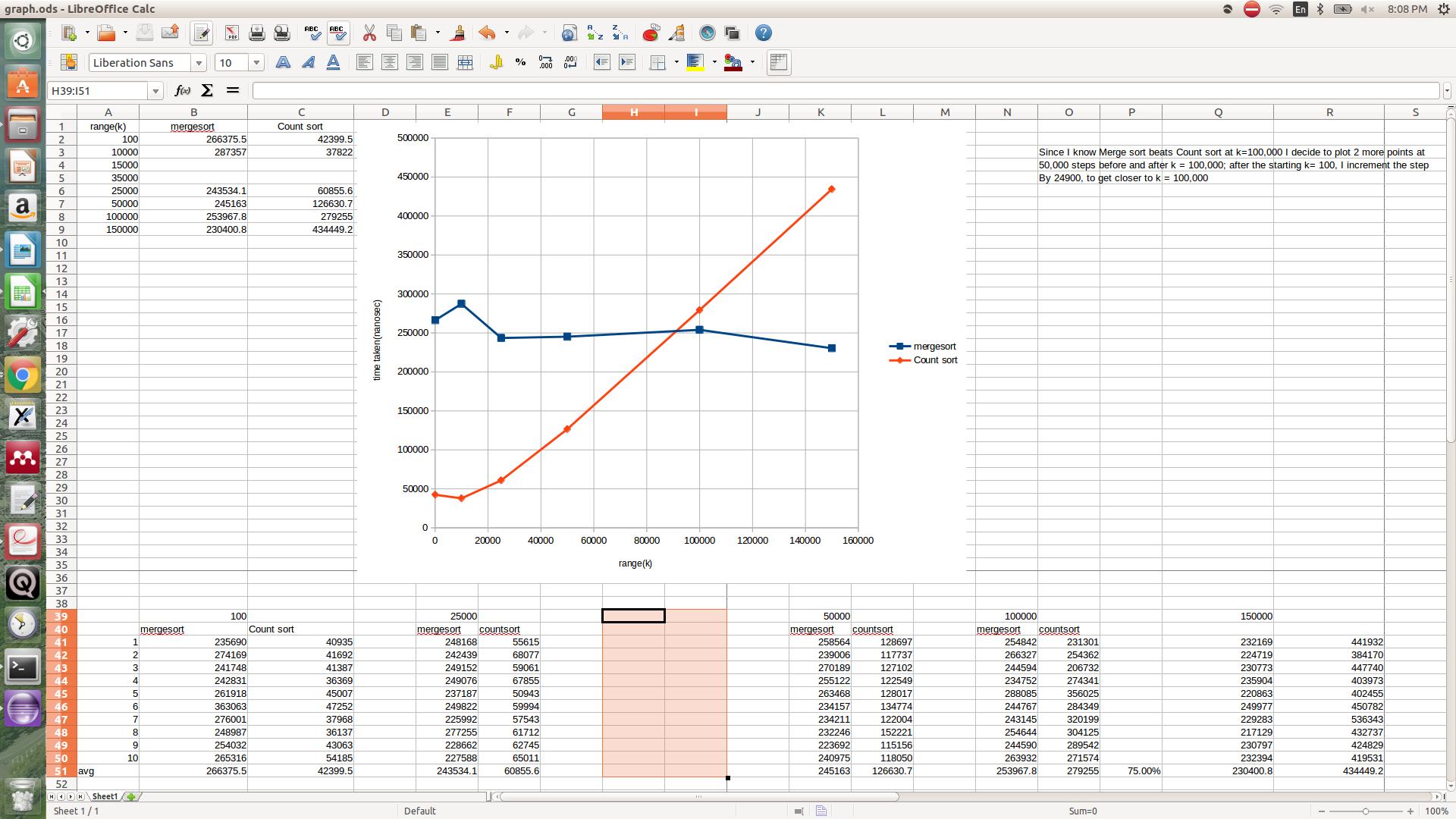
I implemented both Merge Sort and Counting Sort from the website geeksforgeeks.org.

According to the graph, Merge Sort beats Counting Sort around k = 90,000.

Since I know Merge sort beats Count sort at k=100,000 I decide to plot 2 more points at 

50,000 steps before and after k = 100,000; after the starting k= 100, I increment the step

by 24,900 to get closer to k = 100,000.

**package** hw5;

**import** java.util.\*;

**public** **class** Sorting {

**public** **boolean** checkFor02k(**int** arr[], **int** numSize){

**boolean** zero = **false**;

**boolean** k = **false**;

**for** (**int** i=0; i<arr.length; i++){

**if** (arr[i] == 0){

zero = **true**;

}

**if** (arr[i] == numSize){

k = **true**;

}

}

**return** (zero && k);

}

**protected** **class** MergeSort {

**public** **void** merge(**int** numbers [], **int** i, **int** j, **int** k) {

**int** mergedSize = k - i + 1; // Size of merged partition

**int** mergedNumbers [] = **new** **int**[mergedSize]; // Temporary array for merged numbers

**int** mergePos = 0; // Position to insert merged number

**int** leftPos = 0; // Position of elements in left partition

**int** rightPos = 0; // Position of elements in right partition

leftPos = i; // Initialize left partition position

rightPos = j + 1; // Initialize right partition position

// Add smallest element from left or right partition to merged numbers

**while** (leftPos <= j && rightPos <= k) {

**if** (numbers[leftPos] < numbers[rightPos]) {

mergedNumbers[mergePos] = numbers[leftPos];

++leftPos;

}

**else** {

mergedNumbers[mergePos] = numbers[rightPos];

++rightPos;

}

++mergePos;

}

// If left partition is not empty, add remaining elements to merged numbers

**while** (leftPos <= j) {

mergedNumbers[mergePos] = numbers[leftPos];

++leftPos;

++mergePos;

}

// If right partition is not empty, add remaining elements to merged numbers

**while** (rightPos <= k) {

mergedNumbers[mergePos] = numbers[rightPos];

++rightPos;

++mergePos;

}

// Copy merge number back to numbers

**for** (mergePos = 0; mergePos < mergedSize; ++mergePos) {

numbers[i + mergePos] = mergedNumbers[mergePos];

}

}

**public** **void** mergeSort(**int** numbers [], **int** i, **int** k) {

**int** j = 0;

**if** (i < k) {

j = (i + k) / 2; // Find the midpoint in the partition

// Recursively sort left and right partitions

mergeSort(numbers, i, j);

mergeSort(numbers, j + 1, k);

// Merge left and right partition in sorted order

merge(numbers, i, j, k);

}

}

}

// Java implementation of Counting Sort

**protected** **class** CountingSort

{

**void** sort(**int** arr[])

{

//NUMBER SIZE

**int** n = arr.length;

// The output character array that will have sorted arr

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

// Create a count array to store count of inidividul

// characters and initialize count array as 0

**int** count[] = **new** **int**[35001];

//

**for** (**int** i=0; i<35001; ++i)

count[i] = 0;

// store count of each character

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

++count[arr[i]];

}

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

// position of this character in output array

**for** (**int** i=1; i<=35000; ++i){

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

}

// Build the output character array

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

{

output[count[arr[i]]-1] = arr[i];

--count[arr[i]];

}

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

// contains sorted characters

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

arr[i] = output[i];

}

}

}

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

**final** **int** SIZEOFTIMEARRAY = 100;

// create array to store total of 100 runs

**long** time4Merge[] = **new** **long**[SIZEOFTIMEARRAY];

**long** time4Count[] = **new** **long**[SIZEOFTIMEARRAY];

**int** k = 35000;

//create vars to store average of the 2 sorts

**long** avg4Merge = 0;

**long** avg4Count = 0;

Random ranGen = **new** Random();

//initialize sorting object

Sorting toInitializeMergeS = **new** Sorting();

MergeSort MergeS = toInitializeMergeS.**new** MergeSort();

CountingSort CountS = toInitializeMergeS.**new** CountingSort();

**final** **int** NUMBERS\_SIZE = 2000;

**int** numbers [] = **new** **int**[NUMBERS\_SIZE];

**int** numbers1 [] = **new** **int**[NUMBERS\_SIZE];

// FOR 0 TO 1000 RUNS, MAKE NEW INPUTS, GET TIME AND STORE TIME IN ARRAY

**int** i = 0;

**for** (**int** j=0; j<SIZEOFTIMEARRAY; j++){

**for** (i=0; i < NUMBERS\_SIZE; i++){

numbers[i] = ranGen.nextInt(k+1);

}

**for** (i=0; i < NUMBERS\_SIZE; i++){

numbers1[i] = ranGen.nextInt(k+1);

}

// check to make sure there is 0 and k int in array

**while** (!toInitializeMergeS.checkFor02k(numbers, k)){

**for** (i=0; i < NUMBERS\_SIZE; i++){

numbers[i] = ranGen.nextInt(k+1);

}

}

// check to make sure there is 0 and k int in array

**while** (!toInitializeMergeS.checkFor02k(numbers1, k)){

**for** (i=0; i < NUMBERS\_SIZE; i++){

numbers1[i] = ranGen.nextInt(k+1);

}

}

System.*out*.println();

System.*out*.println(j + " UNSORTED: ");

**for** (i = 0; i < NUMBERS\_SIZE; ++i) {

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

}

System.*out*.println();

**for** (i = 0; i < NUMBERS\_SIZE; ++i) {

System.*out*.print(numbers1[i] + " ");

}

System.*out*.println();

System.*out*.println();

/\* initial call to mergesort with index \*/

//stop and record timer for merge in array

**long** startTime = System.*nanoTime*();

MergeS.mergeSort(numbers, 0, NUMBERS\_SIZE - 1);

**long** stopTime = System.*nanoTime*();

time4Merge[j] = stopTime - startTime;

// System.out.println("time for merge " + j + ": " + time4Merge[j]);

//start timer for count

startTime = System.*nanoTime*();

//call to counting sort w index

CountS.sort(numbers1);

//stop and record timer for count in array

stopTime = System.*nanoTime*();

time4Count[j] = stopTime - startTime;

/\*

System.out.println("time for count " + j + ": " + time4Count[j]);

System.out.println( j+" SORTED: ");

for (i = 0; i < NUMBERS\_SIZE; ++i) {

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

}

System.out.println();

for (i = 0; i < NUMBERS\_SIZE; ++i) {

System.out.print(numbers1[i] + " ");

}

System.out.println();

\*/

}

// average the times and print the avg of 100 runs

**for** (**int** j=0; j<SIZEOFTIMEARRAY; j++){

avg4Merge += time4Merge[j];

avg4Count += time4Count[j];

}

avg4Merge /= SIZEOFTIMEARRAY;

avg4Count /= SIZEOFTIMEARRAY;

System.*out*.println("average time of 100000 runs for Merge is: " + avg4Merge);

System.*out*.println("average time of 100000 runs for Count is: " + avg4Count);

**return**;

}

}