Comp 2140 - Data Structures
Lab 4 - Counting Sort & Consistency of Algorithms

‘We are all now connected by the internet like neurons in a giant brain’  Stephen Hawking

Please print your name and your student id in this box. Return this paper (with your answers to the written questions) to the TA at the end of the lab. The TA will comment on your programming assignments, also on this paper.

your name: your id:

The objective of this lab is to explore the counting sort as well as consistency of sorting algorithms.
Please take the following steps in this lab. Download the file Lab4.java. This file contains partial implementations of insertion sort, counting sort, and quick-sort along with methods for initializing arrays randomly and printing them.

1 Take a look at the method public static void testCountingSort(). This method creates an array of size \( n = 10 \), places random integers in the range \([0..5]\) in \( a \), and prints it. Then, it calls countingSort \((a, k)\) and prints the output. Run the code 5 to 10 times. Explain whether the output is always as you expect or not. If not, I) in the box below write an example that you encountered for which the countingSort is not working properly. II) then go through the countingSort method and fix the error. Show the result of your correct test to the TA.

Answer: The output is Not as expected. Here is a possible output:
before counting sort:
5 5 2 1 1 4 0 4 0 5
after counting sort:
0 1 4 5 5 4 0 4 0 5
Note that the output list is not sorted. The problem is that the array \( C \) is initialized with -1 instead of 0. Hence, the computed counts are 1 unit less than the actual counts. To get the full mark, you had to indicate that the answer is not what you expect and fix the code.

2 Take a look at the function public static void testTime(int n, int k). This function creates an array \( a \) of size \( n \), places random integers in range \([0..k]\) in it, and creates a new copy \( b \) of \( a \). Then, it applies Counting Sort and Quick Sort for sorting \( a \) and \( b \) respectively, and prints the time taken by each. In the main function, call testTime four times times. In all calls, let \( n \) be a sufficiently large value (e.g., \( n = 100,000 \)). In the first call, let \( k = 10 \). For other calls, use the following values of \( k \): \( k = 1,000 \), \( k = 100,000 \), \( k = 10,000,000 \). Write the time taken by quick-sort and counting sort algorithms for each test in the space below (you can also use the space in the back page).
Indicate what you observe and justify it in a few sentences. Show the numbers you see to the TA.

Answer: For small values of \( k \), the counting sort has a strong advantage over quick-sort because it runs in linear time while quick-sorts runs in \( O(n \log n) \) on average. As \( k \) increases, the advantage of counting sort vanishes (note that there is also an overhead in assigning a large temporary array \( C \) which matters in experiments like this). For sufficiently large values of \( k \), quick-sort will have an advantage.
Answer:

3 Create a copy of the function testTime and call it testTime2. In the new function, we would like to call and report the time complexity of the insertionSort in addition to QuickSort and countingSort. For that, you need to create a new array c and copy the content of a to c (the same way we copied a to b). You also need to create a copy of the code block in "* * *" which calls the insertionSort(c). Run testTime2 with parameters n = 10,000 and k = 1,000,000 twice. In the first run, the three sorting functions are called with the random array as their input (as before). In the second run, sort the input array before calling the three sorting algorithms (e.g., add quickSort(a) after setRandomly(a, 0, k)). In both cases, show the numbers to the TA. What do you observe and how you justify it?

Answer: Before sorting, quick-sort has advantage while insertion-sort is pretty bad. Note that in this case insertion sort takes $O(n^2)$. After the sorting, quick-sort will be in its worst case (note that we used the first item as the pivot). So, its time deteriorates. The insertion sort however, runs in linear time for sorted input. So, these two algorithms perform quite differently in their best and worst case settings.

Counting sort’s performance might slightly improves for the sorted case because of locality (you keep incrementing the same index of c). But its time complexity is not changed (you don’t need to justify counting-sort behaviour for the full mark)