**Introduction**

In this lab you will compute the running times and RAM usage of the linear sorting algorithm counting sort and compare it to comparison sort algorithms such as merge sort and heap sort. Submit your answers to the questions below in a text file (e.g. Word document). Name your file in name\_surname.docx format. Submit your solution document and Java codes to Canvas.

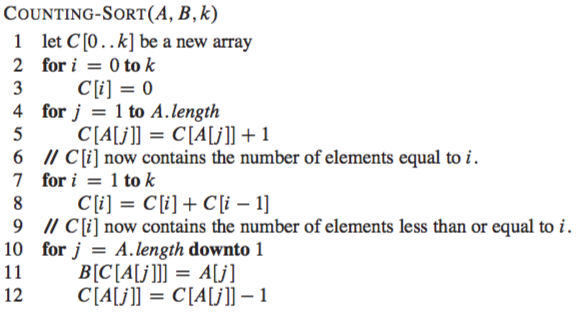
You can use the code templates in linear.java in this lab.

**Problem Statement**

Given an array of integers sort the numbers in this array in ascending order.

**Assignment**

1. (a) Implement a Java method for the counting sort algorithm given below. Be careful with the indices such that indices of array C start from 0 while those of arrays A and B start from 1. You may need to do the necessary adjustments in the indexing of these arrays.



(b) Test your algorithm by choosing an array of size 10. Initialize your array by random numbers from 0 to 9. Make sure your program sorts the array correctly. Include the output of your program for this sample input in your report.

(c) Choose input sizes in the table below, which are multiples of 10, and initialize the values in your array by random numbers from 0 to array\_size-1. Compute the running time of counting sort, merge sort and heap sort in nanoseconds for each of these input sizes and include them to the table below. The codes for merge sort and heap sort are available in the code template. Write a for loop that performs these operations automatically. Do not run them one at a time.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Input size | Input range | Counting sort running time | Merge sort running time | Heap sort running time |
| 10 | 0-9 | **2800** | **18500** | **335400** |
| 100 | 0-99 | **11700** | **121700** | **164000** |
| 1000 | 0-999 | **108800** | **565800** | **565800** |
| 10000 | 0-9999 | **965200** | **2757200** | **2828300** |
| 100000 | 0-99999 | **5830600** | **18600100** | **22027600** |
| 1000000 | 0-999999 | **25871700** | **158859300** | **141015700** |
| 10000000 | 0-9999999 | **923143900** | **1526498500** | **1972810000** |
| 100000000 | 0-99999999 | **14088989000** | **19592602300** | **34262005400** |

Which algorithm performs best at which input size?

Counting sort is faster than other algorithms in all input sizes since its running time is O(n)

(d) Now set the input range to 0-9999999 and fill the table below by repeating the analysis you made in part (c).

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Input size | Input range | Counting sort running time | Merge sort running time | Heap sort running time |
| 10 | 0-9999999 | **36236100** | **15500** | **302100** |
| 100 | 0-9999999 | **33112800** | **109800** | **119300** |
| 1000 | 0-9999999 | **32019300** | **1038000** | **264200** |
| 10000 | 0-9999999 | **13948900** | **2679400** | **2225600** |
| 100000 | 0-9999999 | **24798300** | **18737200** | **21376500** |

Which algorithm performs best at which input size?

Since input size is less than input range, merge sort beats other algorithms in all inputs until 100000.

(e) Set the input size to 100000000 and the input range to 0-9999999. Run count sort, merge sort and heap sort one at a time for this input size. Open a terminal window and type top. Find the processes for the sorting algorithm you executed and record the RAM usage in MEM column. Include the RAM usage of these algorithms into the table below. Compare and comment on the RAM usage of these sorting algorithms.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Input size | Input range | Counting sort RAM | Merge sort RAM | Heap sort RAM |
| 100000000 | 0-9999999 | **2.876.340** | **2.871.476** | **2.803.460** |

Counting sort uses the largest memory, because it is not comparison-based algorithm, it uses the space complexity more than time complexity. After counting, merge sort uses more than heap because it creates sub arrays to sort array.