Technische Universität München

# Informatics I for MSE

**MSE** IN8011
Prof. Bjoern Menze, Giles Tetteh
Philipp Kaeß

**WS 19/20**
Homework 10

**Due Date:** 02.02.20, 23:59

**Release Date:** 24.01.20

## Homework 10: Counting Sort

**Topics:** Sorting, Counting Sort, Complexity, Dynamic Memory Allocation

**Introduction**

In last week's tutorial you were introduced to the concept of Bubblesort, one of the easiest sorting algorithms. We noticed that the complexity of Bubblesort is $O(n^2)$, which means that the time needed to sort a list rises quadratically with an increase in the number of elements to sort $n$. To avoid that expensive increase (especially for larger lists) we introduced Mergesort, which has a significantly better complexity of $O(n\,log(n))$. This is actually the best it can get. No comparison-based sorting algorithm can have a smaller complexity than that. However, in specific circumstances we can avoid the comparisons altogether and achieve an even better complexity.

Given an array of length $n$ filled with integer values in a given range, we iterate over the array to count the number of occurrences of each value, and store these numbers in a histogram. With the help of this histogram we then simply write the values back into the array in the correct order.

Let's look at an example to better understand this procedure. The table below contains $n = 15$ integers in a range of $[-3; 8]$ and should be sorted:

| 6 | 3 | 8 | -1 | 0 | -3 | 6 | 6 | 2 | -3 | 0 | 4 | 6 | 2 | -1 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

We first create a histogram, where we count the occurrences of all values between $-3$ and $8$:

| -3 | -2 | -1 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 2 | 0 | 2 | 2 | 0 | 2 | 1 | 1 | 0 | 4 | 0 | 1 |

In a second pass, we iterate over the histogram and write the respective number of copies into the input array. In this example we would write $-3$ to the first and second spot of the above table, skip $-2$, write $-1$ to the third and fourth spot of the table, etc. After these iterations our input table is sorted:

| -3 | -3 | -1 | -1 | 0 | 0 | 2 | 2 | 3 | 4 | 6 | 6 | 6 | 6 | 8 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

This algorithm is called Countingsort.

Let's take a look at the complexity of our new algorithm. To create the histogram, we iterated over the list once, which therefore had complexity $O(n)$. In the second step we iterated over the histogram, which is as large as our given range of values ($k = max - min + 1$). This step obviously had complexity $O(k)$. In practice we often do not know the maximum and minimum of the values in the input array in advance, so we need an additional pass through the array at the beginning to determine these extrema, which has complexity $O(n)$ as well.

These steps combined yield a complexity of $O(n + k)$, the runtime of Countingsort is linearly dependent on the number of elements $n$ and on the size of the range of the values $k$. Therefore, Countingsort is a very good choice iff (if and only if) the range of values $k$ is not significantly higher than the number of elements to sort $n$.

**Specifications**

The function `counting_sort()` with the input parameters

- `array` , an array of `int`s
- `length` , the length of `array` as `int`

should sort the values in `array` with Countingsort as described above. After a run of the function the values in `array` should be sorted. Please note that the values in `array` can be positive as well as negative, and `length` can be any **non-negative** integer.

**Tasks**

We will provide a template called ***counting_sort.c***. Complete the function `counting_sort()` so that it works as described above. Please note:

- Do not print anything!
- You are provided with the corresponding header file ***counting_sort.h***. It contains the function declaration. We will also provide a main file called ***main_sort.c*** to test your progress. It simply sorts a small, given array with your function and prints the results.
- Take care to only write within the valid range! If you e.g. write in `array` at an invalid index (lower than `0` or higher than `length-1`), you will fail the submission.
- It is possible that you will encounter a situation where you need to create a variable-sized array. Even though it might be possible to do that "statically" (e.g. `int my_array[var_size];`) we highly encourage you to allocate and free the memory dynamically, as the pseudo-static version might not work on your tutor's system. By doing it anyway you will risk failing the submission as well.
- You are allowed to code outside the marked spots and e.g. define own functions, but you are not allowed to include any additional libraries.
- When you are finished, put ***counting_sort.c*** directly in a zip-File named "HW10_Mustermann_Max" (with your name of course) and upload this file to Moodle. Do not upload any other files, and do not put any additional folders into the zip-File.

**Threshold**
To pass this homework, your program has to fulfill the following conditions:

- Your file ***counting_sort.c*** must be compilable and runnable together with the main file without any errors and with the flags -Wall -Werror enabled

Your functions will be tested on a total of 10 cases. To pass this assignment, you have to pass at least 80% (=8) of the test cases.

**Information on Plagiarism and Copyright**

As already announced in the lecture and the tutorials, you are supposed to do this homework alone. This homework is an official part of the exam at the end of this semester, and therefore fraud or plagiarism will be pursued and punished.

You are encouraged to discuss **specific** problems with other students (like e.g. “I don’t know how to print a string, can you help me?”), but you are forbidden to share ideas on the approach/structure of your functions, and of course to show other students your solution. This will result in a similar control-flow and structure of your submissions, which we will be able to detect even if you change variable names or loops etc.

We will compare your submissions with a code-optimized plagiarism scanner and if we find plagiarized code, you will fail this submission. If you are found guilty of plagiarism twice, you will lose the homework bonus, and in severe cases we will take further steps.

**This homework is intellectual property of the TUM, namely the Chair of Image-Based Biomedical Computing. Publishing either this homework task itself or your solutions to it in any form (including on the MSE-Cloud) is strictly forbidden and can be pursued TUM-internally as well as legally.**