# UMass Boston CS 240
# Homework 5
# Due 7/15/2019 11:59 pm

## 1 Counting Sort

You may have learned some sorting algorithms – such as bubble sort and quicksort – in CS 110 and CS 210. This homework is about *counting sort*. Let $n$ be the number of elements to be sorted. Bubble sort and quicksort assume that we can compare any pair of elements and find out, in constant time, which one is larger and which one is smaller. They make no assumption on the values of the elements.

Counting sort assumes the elements are integers between 0 and $m$, and $m$ is much smaller than $n$. For example, let $m$ be 3, and $n$ be 10. Then the elements can be 0, 1, or 2, and there are ten of them. For example,

```
unsigned data[10] = {0, 2, 1, 1, 0, 2, 2, 0, 1, 1};
unsigned count[3];
```

Counting sort uses an array `unsigned count[m]` and initializes all elements in `count` to zero. Then we scan the array `data`. When we see a number `data[i]`, this number is used as the index to the array `count`, and the corresponding count is incremented. After we finish scanning the array `data`, we have `count[0] == 3`, `count[1] == 4`, and `count[2] == 3`. How do we construct the sorted array from these counts? We write 0 for `count[0]` times, 1 for `count[1]` times, and 2 for `count[2]` times.

Bubble sort takes $O(n^2)$ time in the worst case, and quicksort takes $O(n \lg n)$ time on average. Loosely speaking, quicksort is faster than bubble sort. You will learn the big-O notation in CS 310. Counting sort takes $O(m+n)$ time. We say it is a linear time algorithm, which is, loosely speaking, faster than quicksort.

The code in `main.c` is the driver. You implement counting sort in `cntSort.c`. The driver runs your counting sort as well as `qsort()` in the standard library. It verifies that your code works correctly, and reports the runtime of both sorting methods.

## 2 Write a Report

Try all kinds of values for $m$ and $n$ to compare the performance of counting sort and quicksort. Write a report in the file `Report.txt` – make sure the file name is exactly `Report.txt`. Discuss what you have learned from the timing experiments. Note that the values of $m$ and $n$ are in a two-dimensional search space. You probably need a search strategy rather than just wandering

around blindly. You should report at least a pair of $m$ and $n$ when counting sort is faster than quicksort, and another pair when the opposite is the case.

Note that when you try to get the elapsed time of computational experiments, the results heavily depend on the load of the server. You can use the command `htop` to see the running processes and the current load – hit the key 'q' to quit `htop`. When the load is heavy, the timing results are erratic and unreliable, which means if many people try to finish their reports near the deadline, they will mess up each other's timing results, and everyone dislikes everyone else.

Write your code in `cntSort.c`. Compile and run as follows:

```
gcc -Wall main.c cntSort.c -o cntSort
./cntSort -m 64 -n 1048576 -s 2019
```

The m, n, and s are optional command line arguments.

# 3 Grading Rubric

1. (10 points)
   (a) Existence of cntSort.c and Report.txt
2. (40 points) `cntSort.c`
   (a) Sensible pseudo code, variable names, comments, etc.
   (b) `gcc -Wall -c cntSort.c` gives no warnings or errors
   (c) `./cntSort` produces correct output
3. (10 points) `main.c`
   (a) Comments at the beginning of the file
   (b) `gcc -Wall cntSort.c main.c -o cntSort` gives no warnings or errors
4. (40 points) `Report.txt`
   (a) What would the optional command line arguments do?
   (b) A pair of $m$ and $n$ such that counting sort is faster than quicksort
   (c) Another pair such that quicksort is faster than counting sort
   (d) Discussion (at least a paragraph in length): what have you learned from the results?
   (e) Excellent discussion will receive extra credit