Project #1 for extra credit
Miguel Á. Carreira-Perpiñán

# Experimental run times of sorting algorithms

The purpose of this project is to implement several algorithms for sorting arrays, to explore empirically their performance under different situations (algorithm and its parameter values, whether the array is sorted, etc.), and to compare it with theoretical estimates. The more careful your implementation and experiments, and the more insightful your comments about the measured and theoretical run time, the higher your grade. Reread the chapters in the textbook about sorting algorithms (in particular, the sections listed in the course web page). The coding part of the project is a minor extension of some of the labs, so it will be a small part of the grade. What is important (assuming the code is correct) is to understand its performance empirically and in the context of the theoretical estimates.

## Work to do

In this course, we have seen in detail several sorting algorithms: insertion sort, merge sort, heap sort, quicksort, counting sort, radix sort and bucket sort (we also mentioned selection sort, bubble sort, tree sort and possibly others). These algorithms have illustrated different design techniques (iteration, divide-and-conquer, randomization, etc.), and they have different asymptotic running times as a function of the array size $n$ and possibly of other parameters (e.g. the maximum number of different integers $k$ in counting sort), as well as best, worst and average cases. In several of the labs, a bonus question has been to measure the running times of sorting algorithms in a given computer. This project expands on that. It consists of repeating the following for different algorithms (counting sort, in this example):

1. Coding the counting sort algorithm in C.
2. Running your counting sort code with input arrays of different size $n$, for different $k$, and when the input array is in ascending order, descending order, or unsorted. Record the run time $T$ in each case.
3. Plotting a curve $T(n)$ as a function of $n$ for each case (each value of $k$, and for ascending order, descending order and unsorted).

The objective of this exercise is to test whether the formulas predicted by the theory (e.g. "quicksort runs in $\Theta(n^2)$ in the worst case") match well with the measured run time in an actual computer, and if they do not match well, to try to understand why and give an explanation or a guess. (You may be surprised about how well, or how badly, the theory applies.) The more algorithms you study and the more insightful your comments about their measured and theoretical run time, the higher your grade.

Analyze your results and summarize them in a report including, for each algorithm you tested:

- The theoretical run time for the best, worst and average case.
- A plot of the runtime $T$ as a function of $n$ (and possibly other parameters) for ascending order, descending order and unsorted. See an example in fig. 1.
- A concise but insightful discussion of whether the curves match with the theory and an explanation (or educated guess) of why they do not, if they disagree. Given the theoretical and experimental curves, consider questions such as the following. How do the curves compare with the asymptotic running times in the best, worst and average case for the algorithms? How much do the algorithm-dependent, hidden constants in the asymptotic notation matter? What happens for very small or very large $n$ with a computer of finite memory size? How do different parameters affect the result (e.g. different $k$ for counting sort)? When does counting sort beat quicksort? How are the measured run times affected by other processes running in the computer? What happens with multicore or multiple-CPU systems? Etc. Be inquisitive.

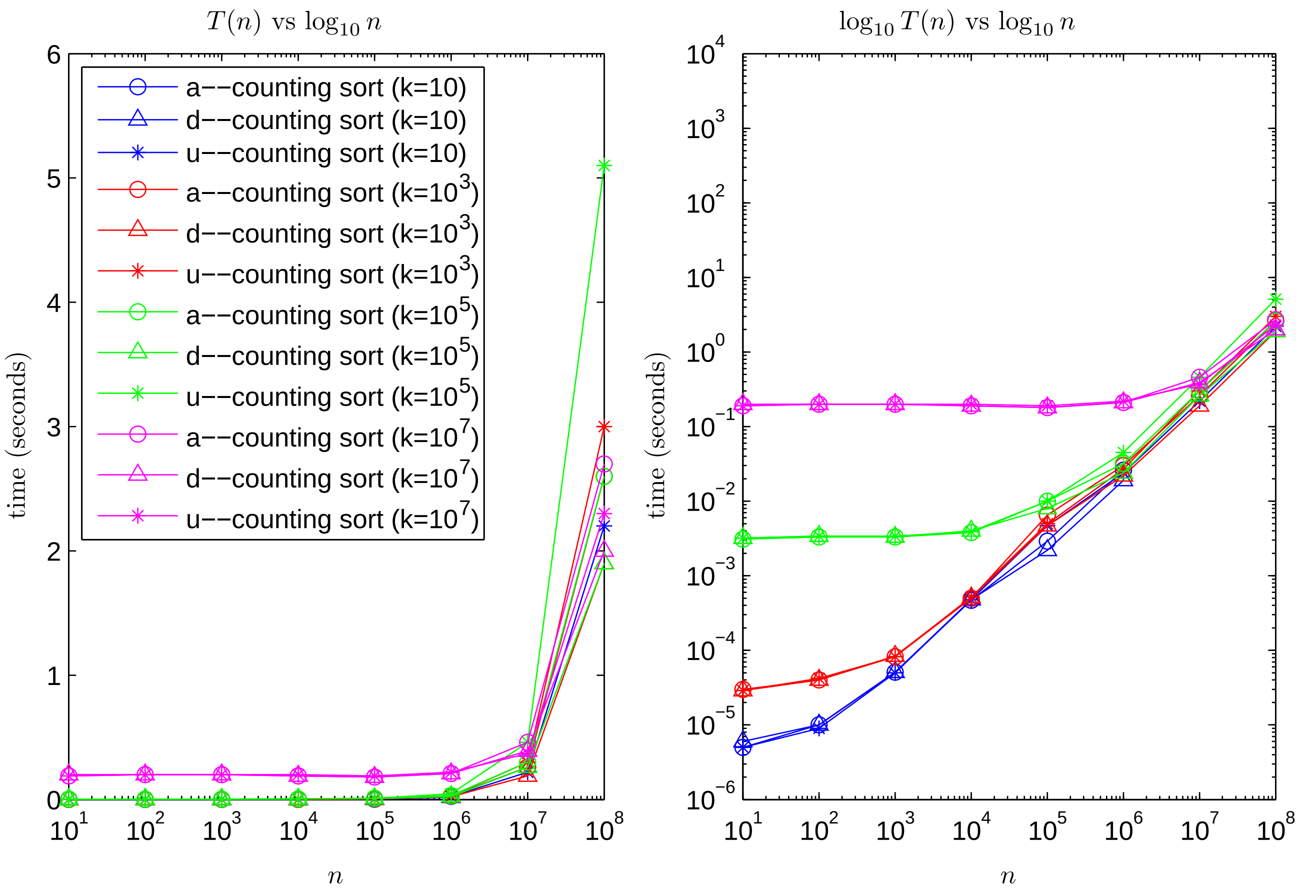

Figure 1: Experimental run times for counting sort for $n \in \{10, 10^2, 10^3, 10^4, 10^5, 10^6, 10^7, 10^8\}$, $k \in \{10, 10^3, 10^5, 10^7\}$ and a random input array of integers that is sorted in ascending order (`'a'`), descending order (`'d'`) or unsorted (`'u'`). *Left*: semilog graph. *Right*: log-log graph.

## What you have to submit

Submit the following (as a single file `cse100.tar.gz`, by email to the TA):

1. A brief explanation of what each group member did.
2. A report `cse100.pdf` of your results, including the plots and a concise but insightful discussion of your analysis.
3. Your C code for the algorithms (each one in a separate subdirectory, e.g. `CountingSort/`). Do not use C++.
4. A single Matlab file `runtimes.mat` containing your measured run times for all the algorithms you tested, from which the plots were generated.
5. If you want, repeat several times the same identical experiment (e.g. counting sort with $n = 1\,000$ and $k = 10$ with a sorted array) to obtain error bars (since no two runs will take the exact same time) and include them in the plot.
6. Give the basic specs for the computer that you used (processor, clock frequency, memory size, number of processors, operating system). **Note**: all run times must be measured with the same computer and (approximately) the same workload. I suggest you run your experiments in a single-core single-CPU system, or that you make sure that the experiments' code runs in a single CPU (see `taskset` in Linux).

## Practical details

For all practical questions, see the TA.

- Generate all your test cases with the C function (provided by us) `rndarray(s,k,n,seed)`, which creates a random array of integers, as follows:
  - `s` (char): array sorted in ascending (`'a'`) or descending order (`'d'`), or unsorted (`'u'`).
  - `k` (int): the array values will be in `0..k`.
  - `n` (int): the size of the array to be generated.
  - `seed` (int): the seed to initialize the pseudorandom number generator. Note: fix this to your student id# for all your experiments.
- Use the C `gettimeofday()` function to measure elapsed time, by bracketing your sorting function with `gettimeofday()` calls. We have provided the template C program `SortTime.c` for this purpose. You just need to merge your sorting function into `SortTime.c` and build the executable. The file `SortTime.c` contains instructions about how to do this.
- Use the Matlab function `PlotRuntime.m` to plot the runtime curves (modify it if you want). For example, if you have 6 curves each with 8 values of $n$, all you need to do is type your running times into a matrix $T$ of $8 \times 6$ (use `NaN` for missing entries) and run `PlotRuntime.m`. The figure will appear on screen and on a file `runtime.eps`.
- You can automate your experiments and save time by running your code with a Unix shellscript that loops over, say, $n = 10^2, 10^3, \dots$ and `s='a','d','u'`. See an example in `sortscript.sh`.
- We suggest you try $n \in \{10^2, 10^3, 10^4, 10^5, 10^6, 10^7, 10^8\}$, for each of `s={'a','d','u'}`; for counting sort, $k \in \{10, 10^3, 10^5, 10^7\}$; for radix sort, $r \in \{1, 5, 10, 20\}$ (the number of bits in each digit).