CS29003 Algorithms Laboratory

Assignment No: 9

Last date of submission: 12–October–2017

A CGPA is technically a floating-point number in the range $[5.00, 10.00]$. However, CGPAs are truncated (or rounded) two places after the decimal point. This means that if x is a CGPA, then $100x$ is an integer in the range $[500, 1000]$. Moreover, $100x - 500$ is an integer in the range $[0, 500]$. In this assignment, you deal with these integer versions of CGPAs. Your task is to sort an array A storing n integer-valued CGPAs.

Part 1: Your first task is to generate a random array A of n CGPAs. In a population of students, the floating-point CGPA x satisfies the following probability distribution: $\Pr[5 \leqslant x \leqslant 6] = 0.05$, $\Pr[6 < x \leqslant 7] = 0.2$, $\Pr[7 < x \leqslant 8] = 0.3$, $\Pr[8 < x \leqslant 9] = 0.3$, and $\Pr[9 < x \leqslant 10] = 0.15$. When we convert x to an integer $X \in [0, 500]$, we have the equivalent probabilities $\Pr[0 \leqslant X \leqslant 100] = 0.05$, $\Pr[100 < X \leqslant 200] = 0.2$, and so on. Assume that in each band, the different CGPA values are equally likely, that is, we have

$$\Pr[X = a] = \begin{cases} 0.05/101 & \text{if } \quad 0 \leqslant a \leqslant 100, \\ 0.20/100 & \text{if } 101 \leqslant a \leqslant 200, \\ 0.30/100 & \text{if } 201 \leqslant a \leqslant 300, \\ 0.30/100 & \text{if } 301 \leqslant a \leqslant 400, \\ 0.15/100 & \text{if } 401 \leqslant a \leqslant 500. \end{cases}$$

You should generate the array A of n CGPAs following this probability distribution. The **`rand()`** library function returns a random integer in the range **`[0,RAND_MAX]`**. This function can be used to generate samples following other probability distributions.

In general, let X be a random variable that assumes integer values in the range $[0, m]$. Let $p_i = \Pr[X = i]$ for $i \in [0, m]$. Consider the cumulative probability $q_i = \sum_{j=0}^{i} p_j$. We generate a (uniformly) random floating-point number $y \in [0, 1]$ by dividing a **`rand()`** output by **`RAND_MAX`**. If $y \leqslant q_0$, we output the integer 0. Otherwise, We locate the $i \in [1, m]$ such that $q_{i-1} < y \leqslant q_i$, and output i. Notice that the cumulative-probability array $(q_0, q_1, q_2, \ldots, q_m)$ is sorted in the ascending order, so the index i can be located by a binary search. The running time of this algorithm to generate the array A is $O(n \log m)$.

In the current situation, a somewhat more efficient ($O(n)$-time) algorithm can be designed. Implement some algorithm to generate the array A of n integer-valued CGPAs following the distribution given above.

Part 2: Implement quick sort in a function *quicksort* for sorting integer arrays. Do not use the **`qsort`** library function defined in the standard C/C++ libraries. Partitioning should be inline, that is, no additional temporary arrays may be used.

Part 3: Write a function *countingsort1* to implement the standard stable counting sort algorithm for CGPA arrays. The function should use a count array C, and a temporary array B of size $n = |A|$ which is finally copied back to the input array A.

Part 4: In this particular application, we are sorting integer values in the range $[0, 500]$ (not records containing these integer values). Therefore the array B can be eliminated altogether. You still need the count array C. Implement this idea in a function *countingsort2*.

The *main*() function

- Read n from the user. For this assignment, n should be in the range $[10^4, 10^8]$.
- Call the function of Part 1 to generate a CGPA array A of size n with elements following the given probability distribution. Make copies of A in two separate arrays B and C.
- Run *quicksort* on A, and report the running time.
- Run *countingsort1* on B, and report the running time.
- Run *countingsort2* on C, and report the running time.

Sample output

```
n = 100000000
+++ Array generation time =   4.904727 sec
+++ Quick sort time       =  11.338450 sec
+++ Counting sort 1 time  =   1.461599 sec
+++ Counting sort 2 time  =   0.748255 sec
```

Submit a single C/C++ source file. Do not use global/static variables.