MIT PRIMES COMPUTER SCIENCE SOLUTIONS

Problem 1: counting sort. In this problem you will explore a non-comparison based sorting using *counting sort*. Your goal is to implement counting sort to sort student records based on expected year of graduation of current students, in increasing order. A student record consists of:

- Last name (a string of no more than 30 English letters and symbols, such as a hyphen, single quotation mark, and space), followed by a comma
- First name (a string of no more than 30 English letters and symbols, such as a hyphen, single quotation mark, and space),
- GPA (a number between 0.00 and 4.00 with two decimal places),
- The year of graduation (a number between 2018 and 2022, inclusive).

For instance, a (small) valid input data set is:

```
5
Lee, Riley 3.34 2020
Lopez, Maria 3.41 2019
Martin, Claire-Marie 2.78 2020
O'Neil, Benjamin 3.62 2021
Smith, Jordan 3.07 2018
```

Note that sorting by graduation year must preserve the original order for students with the same graduation year. Thus after sorting by graduation year your program must output:

```
Smith, Jordan 3.07 2018
Lopez, Maria 3.41 2019
Lee, Riley 3.34 2020
Martin, Claire-Marie 2.78 2020
O'Neil, Benjamin 3.62 2021
```

Run your program on a dataset of at least 100,000 elements (randomly generate nonsensical names and GPAs) to get the baseline for subsequent modifications.

Solution. Run CS1.java. More information can be found within the source file.

Problem 2: sorting by year of graduation and GPA. Now suppose you are given the same kind of data, but you need to sort it by GPA in decreasing order within each group. If two students have the same graduation year and GPA, their order in the input file must be preserved. You may use a combination of two passes of sorting: one by year of graduation, and one by GPA. Alternatively you can use a one-pass approach using whatever sorting algorithm you like to sort by both fields at once.

Your goal is to make your sorting fast; use the timer in the program to measure elapsed time (see general requirements for details). Please answer the following about your program:

- What approaches have you considered or tried, and why did you pick the approach that you are submitting?
- What is the worst case, the best case, and (approximately) the average case as Big-O of the size of the data set? Explain your answer, use the known efficiencies of the sorting that you are using.
- Assuming that you are sorting N elements, how much memory does your approach use? Please explain what the memory is used for.

Solution. Run CS2.java. More information can be found within the source file.

I wrote this program after CS1.java, so I knew that counting sort is efficient for a problem like this, since the memory requirement is minimal and the size of the counting arrays are small. Therefore, I applied the code I had already written to sort the students by graduation year. In order to then sort by GPA within each graduation year, I initially tried to apply a heap sort, but after noticing a 60 millisecond increase in speed by switching to counting sort, I decided that it would be most efficient to perform a counting sort on the main array for the intervals where the graduation years were the same. Also, I had to make the GPA a 3 digit integer in order to make it possible to use a counting sort.

Since the sorting is purely for-loops and not recursive, the best, worst, and average cases are all the same. Given that the program runs a counting sort once to arrange by graduation year ($O(N + K)$), then from 1 to 5 times to sort by GPA within each graduation year $O(5(N + K))$, we can write $O(6(N + K))$, which simplifies to $O(N + K)$.

The memory used in a counting sort is $O(N + K)$ as well, since there are two arrays of size N (input and output) and one array of size K (count array). Thus, using the same logic as before, the memory used is $O(N + K)$.

Problem 3: sorting by student IDs and year of graduation. Suppose the data set is changed to add unique 9-digit student IDs:

```
5
345678916 Lee, Riley 3.34 2020
342368905 Lopez, Maria 3.41 2019
467231450 Martin, Claire-Marie 2.78 2020
398765468 O'Neil, Benjamin 3.62 2021
602345671 Smith, Jordan 3.07 2018
```

(1) Is it possible to use counting sort to sort students by IDs? Please clearly explain your answer.

Solution. The only way counting sort could not be used to sort students by ID would be if the count array did not support 9 digit numbers. However, in Java, the maximum supported array size is $2^{31} - 1$. The 9 digit threshold lies between 2^{29} and 2^{30}, so it is theoretically possible to use counting sort. It would require a massive amount of memory. More specifically, in Java, the counting array would require 4GB ($4 * 10^9$) of memory, which is unreasonable for a program which should use far less.

(2) Describe how you would use radix sort to sort students by IDs by having a separate sorting pass through the data set for each digit. Would this be a good approach? Please explain your answer; address both memory and time costs. What is the Big-O efficiency of this approach?

Solution. I would perform 9 counting sorts on the data, each time with a different digit being compared. This approach would create 9 sets of 2 arrays with size N and 1 array with size 10, and would iterate through a counting sort in which it would iterate through the data $2N + 10$ times. Thus, the time efficiency can be simplified from $O(9(2N + 10))$ to $O(N)$.

(3) Now suppose that we use radix perform only three passes through the data set, one for each group of three digits. Each group of three digits has 1,000 possible values. What are the tradeoffs between this method and the one in the previous question? What is the big-O efficiency of this method? Which of the two would be more efficient in practice? Please explain your answer.

Solution. Since $3 \cdot 1000 > 9 \cdot 10$ (number of iterations times count array size), this method has more efficient memory usage if $N \geq 243$, since:

$$3(2N + 1000) \leq 9(2N + 10) \Rightarrow 6N + 3000 \leq 18N + 90 \Rightarrow 12N \geq 2910$$

$$\therefore N \geq 242.5$$

Because the time costs are the same as the memory usages, this method is also more time efficient if $N \geq 243$. However, although the two methods are practically different, their big-O efficiencies are both $O(N)$. In practice, most likely more than 243 students are being sorted, so this method would be more efficient.

(4) Write a program to sort the data by IDs. You may use radix sort as described above or in a different way; you may also use a different method altogether if you think it would run faster than radix sort.

Solution. Run CS3_4.java. More information can be found within the source file.

(5) Describe what you have tried and explain your choice.

Solution. After looking at it theoretically and also measuring the general duration of both radix sort methods presented in the above subproblems, I came to the conclusion that the latter of the two was more efficient for my purposes, thus explaining why I chose to implement it as my sorting method for this problem.